7.0

OBJECTWINDOWS™

PROGRAMMING GUIDE

- TUTORIAL
- USING OBJECTWINDOWS
- REFERENCE

BORLAND

ObjectWindows™

Programming Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

PRINTED IN THE USA.
10 9 8 7 6 5 4 3 2 1

R1

C O N T E N T S

Part 3 Advanced ObjectWindows

T A B L E S

F I G U R E S

L I S T I N G S

Windows without pain

This book contains complete documentation for ObjectWindows, the object-oriented application framework for Microsoft Windows. It describes not only what ObjectWindows can do and how, but also why. You'll find that ObjectWindows is by far the fastest way to build sophisticated, powerful Windows applications.

Although this book explains much of how Windows works and how your applications interact with it, we won't try to cover all the aspects of programming the Windows Application Programming Interface (API). The are numerous books, larger than this one, that try to explain how to program Windows on its own terms. ObjectWindows insulates you from many of the tedious and arcane tasks involved in writing traditional Windows applications. You still have access to all the features available from Windows, but you only deal with Windows when you want to. The rest of the time, you let ObjectWindows handle it for you.

What's new in ObjectWindows?

This version of ObjectWindows adds new objects to the hierarchy and adds some new capabilities to existing objects. Changes to the existing objects are backward-compatible, so existing Object-Windows code should compile with only minimal changes.

This version of ObjectWindows has the following new features:

- Support for data validation (see Chapter 13)

- Objects for printing documents and window contents (see Chapter 15)
- Borland Windows Custom Controls
- Multiple units

In addition, this manual includes the following new material:

- Expanded tutorial
- New chapter on Windows messages (Chapter 16)
- Reorganized chapters on the objects in the hierarchy
- More complete inheritance information in the reference section

Why ObjectWindows?

Windows introduces many new wrinkles you might never have had to think about before, like dealing with text and graphics in resizable windows, interacting with other programs in a multitasking environment, and manipulating the nearly 600 functions in the Windows API. Perhaps the most frustrating part can be figuring out just which basic things your program needs to do to function in Windows, and then making sure you've done all of them.

It requires a lot from an application just to qualify as a Windows program. For example, it can't write directly to the screen or directly modify memory. In addition, a Windows application must know how to respond to the volley of notification messages Windows sends to its applications in response to user events like selecting a menu option.

But we wouldn't leave you to deal with that all by yourself. We give you a head start: ObjectWindows.

ObjectWindows is an object-oriented library that encapsulates most of the behaviors that every window has to know and lets you inherit all that instead of reinventing it every time you start a new program. By providing you with that stable, solid framework, we enable you to focus on the parts of the program you *want* to write, instead of the parts that are common to all Windows applications.

What you need to know

There are several things you need to know before you start writing applications for Windows. First is that you need to know how to *use* both Pascal and Windows. The elements of programming in Pascal can be found in the *User's Guide* and the *Language Guide*, and information on using Windows comes with the Windows software.

In addition, you need to be comfortable with object-oriented programming in order to use ObjectWindows. Applications written with ObjectWindows make extensive use of object-oriented techniques, including inheritance and polymorphism. These topics are covered in the chapter "Object-oriented programming," in the *User's Guide*.

In addition to object-oriented techniques, you also need to be familiar with the use of pointers and dynamic variables, because nearly all of ObjectWindows' object instances are dynamically allocated on the heap. Pointers are also explained in the *User's Guide*.

How to use this book

The ObjectWindows Programming Guide is expanded to make it more complete and easier to use. If you're already familiar with ObjectWindows, you'll probably want to skim chapters 7, 13, 15, and 16 to see what's new. If you're new to ObjectWindows, you should read through all of Part 1, "Learning ObjectWindows." The tutorial walks you through building a complete ObjectWindows application, explaining the principles of ObjectWindows along the way.

What's in this book?

Because ObjectWindows is a new approach to Windows programming, and because it uses some techniques you might not be familiar with, this manual includes explanatory material. Complete reference material for ObjectWindows can be found in Chapter 21, "ObjectWindows reference."

This manual is divided into four parts:

- Part 1, *Learning ObjectWindows*, introduces you to the principles of writing Windows applications with ObjectWindows, including a tutorial that walks you through the process of writing and extending an ObjectWindows application.
- Part 2, *Using ObjectWindows*, gives greater detail on the elements of ObjectWindows itself, including an overview of the object hierarchy and how it interacts with the Windows environment and detailed explanations of the parts of the hierarchy and how they are used.
- Part 3, *Advanced ObjectWindows*, discusses important topics for more advanced programming for Windows, especially the parts where you have to interact with the Windows environment directly, including Windows messages, graphics, and use of resources. There are also chapters on ObjectWindows' collection and stream objects.
- Part 4, *ObjectWindows reference*, is a complete reference lookup for all the objects and other elements included in the Object-Windows units.

PART 1

Learning ObjectWindows

C H A P T E R

1

Stepping through Windows

In the next few chapters, you will build a graphical, interactive Windows program, complete with menus, file saving and loading, graphics and text drawing, and custom controls. On the way, you will be introduced to the major principles of Windows application design, such as message processing, managing parent and child windows, and automatic graphics redrawing.

This walk-through consists of twelve steps:

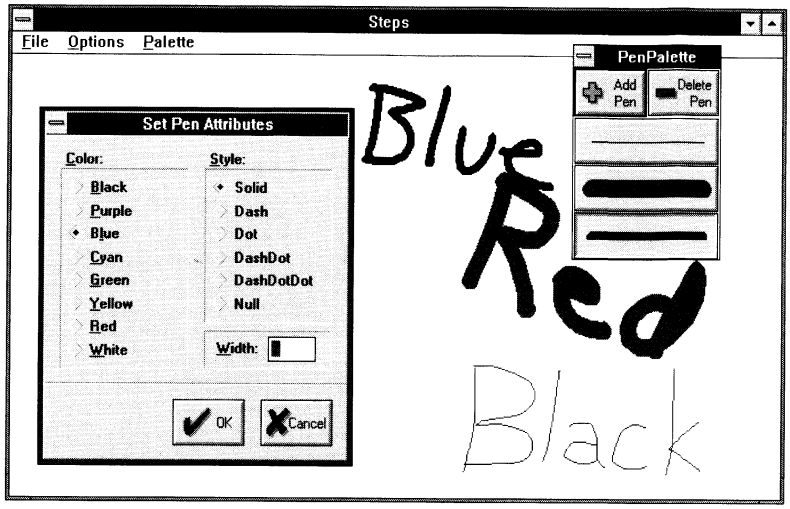
- Step 1: Creating a basic application
- Step 2: Drawing text in the window
- Step 3: Drawing lines in the window
- Step 4: Adding a menu bar
- Step 5: Adding a dialog box
- Step 6: Changing pen attributes
- Step 7: Redisplaying graphics
- Step 8: Storing the drawing in a file
- Step 9: Printing the image
- Step 10: Adding a pop-up window
- Step 11: Adding custom controls
- Step 12: Creating a custom window control

The source code for the application is provided at various stages on the distribution disks. The files are named STEP01.PAS, STEP02.PAS, and so on, corresponding to the steps in the tutorial.

In some cases, there are intermediate steps on disk as well. These are named STEP01A.PAS, STEP01B.PAS, and so on.

Figure 1.1 shows the application you create by the end of this tutorial.

Figure 1.1
An ObjectWindows application



Following Step 12 on page 85 there are suggestions for extending the program.

Step 1: Creating a basic application

```
Step 1:  Basic App
Step 2:  Text
Step 3:  Lines
Step 4:  Menu
Step 5:  About Box
Step 6:  Pens
Step 7:  Painting
Step 8:  Streams
Step 9:  Printing
Step 10: Palette
Step 11: BWCC
Step 12: Custom ctls
```

You start your application development by writing a bare-bones ObjectWindows application, called *Steps*. This program, found in STEP01A.PAS, can serve as the starting point for all of the programs you write in ObjectWindows. *Steps* instantiates and creates the application's main window.

All ObjectWindows programs must use the unit *OWindows*. *OWindows* contains the standard objects used by ObjectWindows for applications and windows. Most Windows applications also include dialog boxes and their associated controls. ObjectWindows provides objects for these in the *ODialogs* unit. Objects related to printing are in the *OPrinter* unit. Programs that use collections and streams need to use the unit *Objects*.

In addition to the ObjectWindows units, most programs need to use *WinTypes* and *WinProcs*. These two units define types and

For an overview of what's in the ObjectWindows units, see Chapter 7, "The ObjectWindows hierarchy."

constants and procedures and functions, respectively, that make up the Windows Application Programming Interface (API). Applications that take advantage of features added in Windows versions after 3.0 need to use other units in addition to *WinTypes* and *WinProcs*.

Application requirements

All Windows programs have a main window that appears when the user starts the program. The user quits the application by closing the main window. In an ObjectWindows application, the main window is a *window object*. This object is owned by the *application object*, which is responsible for creating and displaying the main window, processing Windows messages, and terminating the application. The application object acts as an object-oriented surrogate for the application itself. In the same way, ObjectWindows provides window, dialog box, and other object types to hide the details of Windows programming.

Every ObjectWindows program must define a new application type that descends from the supplied type, *TApplication*. In *Steps*, this type is called *TMyApplication*. Here is the main block of *Steps*:

```
var
  MyApp: TMyApplication;
begin
  MyApp.Init('Steps');
  MyApp.Run;
  MyApp.Done;
end.
```

Init is *TMyApplication*'s constructor and creates the new application object, *MyApp*. It also sets the application's name (an object field) to 'Steps' and creates and shows the application's main window. *Run* starts a series of method calls that sets the new Windows application in motion. *Done* is *TMyApplication*'s destructor.

Defining the application type

Your application must derive a new type from the standard ObjectWindows type *TApplication* (or some type derived from *TApplication*). This new type should override at least one method, *InitMainWindow*. *TApplication.InitMainWindow* is called automatically by ObjectWindows to set up your program's main window. Every ObjectWindows application *must* construct its

Application objects are described in detail in Chapter 8.

own main window to do anything meaningful.

Here's the definition of the type *TMyApplication*:

```
type
  TMyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;
```

Initializing the main window

InitMainWindow is responsible for constructing the window object that will serve as the application's main window. This main window object is stored in the application object's *MainWindow* field. The application object *owns* the main window object, but the two are not related in the inheritance hierarchy.

```
procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil, 'Steps'));
end;
```

You will usually modify the *InitMainWindow* method to supply a new main window type. The above method uses an object instance of *TWindow*, an ObjectWindows-supplied window type which defines the most generic window. In Step 2 (page 19), you will replace it with a more interesting window type.

At this point, *Steps* does nothing but display a blank window that can be moved, resized, maximized, minimized and closed. Listing 1.1 shows a full listing of *Steps* up to this point.

Listing 1.1
STEP01A.PAS produces a captioned main window.

```
program Steps;

uses OWindows;

type
  TMyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil, 'Steps'));
end;

var MyApp: TMyApplication;
begin
  MyApp.Init('Steps');
  MyApp.Run;
  MyApp.Done;
end.
```

The main window object

Main windows are discussed in detail in Chapter 8, "Application objects."

So far, *Steps* consists of two objects: an application object and a window object. The application object, *MyApp*, is an instance of *TMyApplication*, the type you derived from *TApplication*. The window object, held in the *MainWindow* field of *MyApp*, is an instance of *TWindow*, a generic ObjectWindows window. In all but the simplest programs, you need to define your own window type for the main window, incorporating application-specific behavior. In this section, you bring the main window object to life by deriving a more specialized window type from *TWindow*.

What is a window object?

An application object encapsulates the standard behaviors of a Windows application, including construction of the main window. The *TApplication* type provides the fundamental behavior of every application you create.

Window objects are discussed in detail in Chapter 10, "Window objects."

Similarly, a window object encapsulates the behavior of the windows that ObjectWindows applications create, including their main windows. These behaviors include being displayed, resized, and closed; responding to user events such as clicking, dragging, and choosing menu options; and displaying controls, such as list boxes and buttons. Type *TWindow* and its ancestor *TWindowsObject* provide the methods and fields for this basic window behavior.

In order to make your programs useful and interesting, you have to create new window types derived from *TWindow*. The new types inherit *TWindow*'s fields and methods and add some of their own. Overall, the object-oriented approach saves you from constantly "reinventing the window."

Handles

All window objects have at least three fields: *HWindow*, *Parent*, and *ChildList*. *HWindow* holds the handle to the window. A handle is a unique number that associates an interface object, such as a window, dialog box, or control object, with its corresponding element on the screen.

For details about window handles and when you can use them, see Chapter 10, "Window objects."

Thus, *HWindow* holds an integer that identifies the appropriate screen element. It's a lot like a claim number at a coat check. Just as you present a claim check to get your coat, you present your handle to get your window. In most of your work with window objects, you do not have to manipulate the window handle

directly, but you need it when calling Windows functions directly. For example, in this step, you will call the *MessageBox* function. *MessageBox* requires that you supply a parameter identifying the message box's parent window. You supply the main window whose handle is stored in its *HWindow* field:

```
MessageBox(MainWindow^.HWindow, 'Do you want to save?',
  'File has changed', mb_YesNo or mb_IconQuestion);
```

Parents and children

Most applications use multiple windows, and these need to be linked together so they can act in concert. For example, when you terminate an application, the application must have some way of cleaning up all the windows it is responsible for. In general, Windows handles this by linking windows as parents and children. A parent window is responsible for its children. ObjectWindows provides fields for each window object to keep track of its parent and any number of children.

Parent-child window relationships are described in detail in Chapter 9, "Interface objects."

The *Parent* field holds a pointer to the window's parent window object. This is not a parent as in an ancestor type, but more like an owner window. Parent window relationships are described in Step 10 (page 68).

The third window object field is *ChildList*, which holds a linked list of the window's *child windows*, if any. You'll add child windows to your program in Step 10 (page 68).

Creating a new window type

Now that you have some idea of what a window object holds, you can derive a new window type from type *TWindow* to serve as a main window for *Steps*. First, update the type definitions to specify the new type *TStepWindow*. Also, be sure to define a pointer to the *TStepWindow* type, *PStepWindow*, which will be useful when you instantiate *TStepWindow* objects.

```
type
  PStepWindow = ^TStepWindow;
  TStepWindow = object(TWindow)
  end;
```

Then update *TMyApplication.InitMainWindow* so it creates a *TStepWindow* rather than a *TWindow* as its main window.

```
procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PStepWindow, Init(nil, 'Steps'));
end;
```

Defining the new type and instantiating it in *InitMainWindow* is all that's required to define a new type for *TMyProgam*'s main window. The application object calls methods to create the window interface element (*Create*) and to display it on the screen (*Show*). You will almost never use these methods directly. Normally, they are called for you when you call the application object's *MakeWindow* method.

MakeWindow is explained in Chapter 9, "Interface objects."

However, *TStepWindow* defines no new behaviors beyond those inherited from *TWindow* and *TWindowsObject*. In other words, it doesn't make *Steps* any more interesting. In the next section, you'll add some behaviors.

Responding to messages

The quickest way to make a window object useful is to make it respond to some Windows messages. For example, when you click the left mouse button in the main window of *Steps*, Windows sends a *wm_LButtonDown* message to the window, which ObjectWindows intercepts and sends to the corresponding window object. This tells the window object that the user clicked the mouse in it. It also passes the coordinates of the point where the user clicked. This information will be used in Step 2 of this tutorial (page 19).

Windows messages are defined in the WinTypes unit.

Similarly, when the user clicks the right mouse button, the main window object receives the *wm_RButtonDown* message sent by Windows. The next step is to teach the main window, an instance of *TStepWindow*, how to respond to these messages and do something useful.

To intercept and respond to Windows messages, you define a method in your window object for each type of incoming message you want to respond to. These are called *message response methods*. To mark a method definition header as a response method, you add an extension to the virtual method, which is the identifier of the message it responds to. For example, the method defined here responds to all *wm_LButtonDown* messages:

All programs and units that redefine message-response methods must use WinTypes.

```
type
  TStepWindow = object(TWindow)
    procedure WMLButtonDown(var Msg: TMessage); virtual wm_First +
      wm_LButtonDown;
  end;
```

All messages in Windows, including Windows messages and menu commands, are represented as numbers. But every message-response method has to have a unique number, so that different methods are called for Windows messages and commands that happen to have the same numbers.

To make this easy for you, ObjectWindows defines constants for each kind of message: *wm_First* for windows messages, *cm_First* for command messages, and *nf_First* for notification messages. These constants are explained in more detail in Chapter 7, but for now, just remember that when you write a method to respond to a message that starts with *wm_*, you add *wm_First* to it.

Msg is a record of type *TMessage* that holds information such as the coordinates of the point that was clicked. All message-response methods *must* take a single variable parameter of *TMessage* type. You'll examine the *Msg* argument further in Step 2 (page 19).

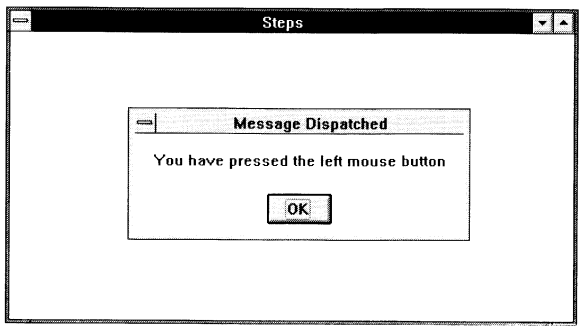
Programs that call MessageBox or other Windows API functions must use WinProcs.

For now, you can just define response methods that display message boxes announcing that the mouse buttons have been pressed. Later, you'll add more useful responses. Here is the definition for the left button response method:

This completes STEP01B.PAS.

```
procedure TStepWindow.WMLButtonDown(var Msg: TMessage);
begin
  MessageBox(HWindow, 'You have pressed the left mouse button',
    'Message Dispatched', mb_OK);
end;
```

Figure 1.2
Steps responding to a user event



The full source code for this step is listed in STEP01B.PAS.

Terminating an application

Steps terminates when the user double-clicks on the Control-menu box of its main window, the small square in its upper-left corner. The window and the application close immediately. This behavior is fine for simple programs but can cause trouble in more complex cases.

A good application always asks if the user wants to save unsaved work before quitting. You can easily add this behavior to your ObjectWindows applications. Start with *Steps* and add the ability to double-check the user's request to quit.

When the user tries to close your ObjectWindows application, Windows sends a *wm_Close* message to the main window, which calls the application object's *CanClose* method. *CanClose* is a Boolean function that indicates whether it is OK (*True*) to close the application. As a default, the *CanClose* method inherited from *TApplication* calls the *CanClose* method of the main window object. In most cases, it is the main window object that decides if it is OK to close.

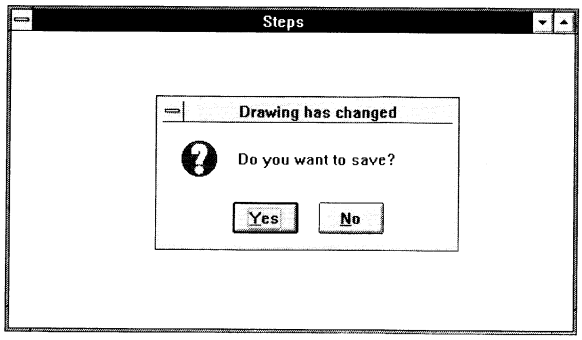
Redefining CanClose

Your main window type, *TStepWindow*, inherits a *CanClose* method from *TWindowsObject* that calls the *CanClose* methods of each of its child windows, if any. If there are no child windows (as in this case), *CanClose* simply returns *True*. To modify an application's closing behavior, you redefine a *CanClose* method for your main window object type:

```
function TStepWindow.CanClose: Boolean;
var Reply: Integer;
begin
  CanClose := True;
  Reply := MessageBox(HWindow, 'Do you want to save?', 'Drawing has
    changed', mb_YesNo or mb_IconQuestion);
  if Reply = id_Yes then CanClose := False;
end;
```

Now, when users try to close *Steps*, they are presented with a message box that asks, "Do you want to save?" Clicking Yes causes *CanClose* to return *False* and prevents the main window and application from closing. Clicking No returns *True* and the application terminates. In Step 8 (page 59) you'll give this message box some meaning. Figure 1.3 shows the modified *Steps*.

Figure 1.3
Steps with refined closing behavior



Further refining closing

Of course, a message telling you that your drawing has changed is only useful if the program actually detects that the drawing has changed. By adding a Boolean field to *TStepWindow*, you can set a flag when the drawing actually changes and only display the message box when the flag is set.

Remember that when you add the field, you also need to initialize that field, so override *TStepWindow*'s constructor:

```
type
  PStepWindow = ^TStepWindow;
  TStepWindow = object(TWindow)
    HasChanged: Boolean;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    :
  end;

constructor TStepWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  HasChanged := False;
end;
```

Next, change the *CanClose* method to check *HasChanged* before displaying a message box:

This brings you to STEP01C.PAS.

```
function TStepWindow.CanClose: Boolean;
var Reply: Integer;
begin
  CanClose := True;
  if HasChanged then
  begin
    Reply := MessageBox(HWindow, 'Do you want to save?',
      'Drawing has changed', mb_YesNo or mb_IconQuestion);
```

```
    if Reply = id_Yes then CanClose := False;
  end;
end;
```

Later, when you actually do change the drawing, you'll set *HasChanged* to *True*. Listing 1.2 shows the full source code to *Steps* thus far.

Listing 1.2
This is STEP01C.PAS.

```
program Steps;

uses WinTypes, WinProcs, OWindows;

type
  TMyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

type
  PStepWindow = ^TStepWindow;
  TStepWindow = object(TWindow)
    HasChanged: Boolean;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    function CanClose: Boolean; virtual;
    procedure WMLButtonDown(var Msg: TMessage);
      virtual wm_First + wm_LButtonDown;
    procedure WMRButtonDown(var Msg: TMessage);
      virtual wm_First + wm_RButtonDown;
  end;

constructor TStepWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  HasChanged := False;
end;

function TStepWindow.CanClose: Boolean;
var Reply: Integer;
begin
  if HasChanged then
  begin
    CanClose := True;
    Reply := MessageBox(HWindow, 'Do you want to save?',
      'Drawing has changed', mb_YesNo or mb_IconQuestion);
    if Reply = id_Yes then CanClose := False;
  end;
end;

procedure TStepWindow.WMLButtonDown(var Msg: TMessage);
begin
  MessageBox(HWindow, 'You have pressed the left mouse button',
    'Message Dispatched', mb_Ok);
end;
```

```
procedure TStepWindow.WMRButtonDown(var Msg: TMessage);
begin
  MessageBox(HWindow, 'You have pressed the right mouse button',
    'Message Dispatched', mb_Ok);
end;

procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PStepWindow, Init(nil, 'Steps'));
end;

var MyApp: TMyApplication;

begin
  MyApp.Init('Steps');
  MyApp.Run;
  MyApp.Done;
end.
```

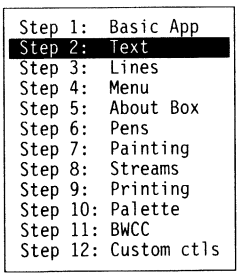
C H A P T E R

2

Filling in the window

Your program doesn't do anything very interesting yet. In this chapter, you'll take *Steps*, which is no more than a shell of a program, and transform it into a useful, interactive graphics application. First, you'll draw text on *Steps*' main window. Then you'll transform *Steps* into a full graphics application that lets you draw lines of varying widths into the main window.

Step 2: Drawing text in the window



As a first step toward creating a drawing program, draw some text characters on your window. Specifically, you draw the text in response to the left mouse button clicks you originally intercepted in Step 1 (page 8). But instead of bringing up a message box, this time you respond by drawing text that shows the coordinates of the point where you clicked on the window.

In a graphical environment like Windows, text is *drawn* as graphics instead of being written as characters to a standard output device. In some ways, this is similar to using cursor-positioning in the *Crt* unit in DOS programs, where you specify the location of each text element, rather than relying on the scrolling of the screen. In Windows, of course, you also have control over the size, style, typeface, and color of your text.

Drawing on a display context

Display contexts are described in detail in Chapter 17, "The Graphics Device Interface."

To draw text in the main window of *Steps*, you need to have something to draw *on*. Windows have special features that handle their graphics, called display contexts. Think of a display context as an element that represents the drawing surface of a window. A display context is required by Windows for drawing any text or graphics in a window.

What is a display context?

The display context serves three important drawing functions.

- It ensures that you do not draw your text and graphics outside the surface of a window.
- It manages the selection and deletion of drawing tools: pens, brushes, and fonts. Step 3 (page 23) shows an example of selecting a new pen tool, but first start with drawing text.
- It provides device independence. Your program uses standard Windows API functions to draw on the display context, and then a device renders the image on itself. In Step 9 (page 63), you'll see how the same commands you use to draw in a window can also draw on a printer.

Windows manages display contexts in its own memory space, but your application can keep track of display contexts using handles. Like a window handle, a display context handle is a number that identifies the correct Windows display context.

Since you need a display context to draw in your window while you drag the mouse, create a new field in your main window object called *DragDC* to hold the handle to the display context. *DragDC* is of type *HDC*, which is equivalent to the type *Word*.

To use a display context, your program must

- Obtain a display context
- Draw on it
- Release the display context

Obtaining a display context

In order to draw on a window, you must first obtain a display context. Do this by calling the Windows function *GetDC* from within one of the window type's methods, right before drawing to the screen:

```
DragDC := GetDC(HWindow);
```

Using a display context

Now you can use *DragDC* as a parameter in Windows graphics function calls that require a handle to a display context. Here are a few examples:

```
TextOut(DragDC, 20, 20, 'Sample Text', 11);
LineTo(DragDC, 30, 45);
```

Releasing a display context

After drawing text or graphics, you *must* release the display context as soon as you finish drawing.

```
ReleaseDC(HWindow, DragDC);
```

If you do not release all obtained display contexts, you will soon run out of them and your application will fail, usually causing your computer to hang. If your application fails the third or fourth time you draw something, check for unreleased display contexts.

GDI and its memory use are explained in Chapter 17, "The Graphics Device Interface."

Windows allocates five display contexts per application, which are shared through *GetDC*. You can create more with *CreateDC*, as long as Windows has enough memory set aside.

Windows coordinates

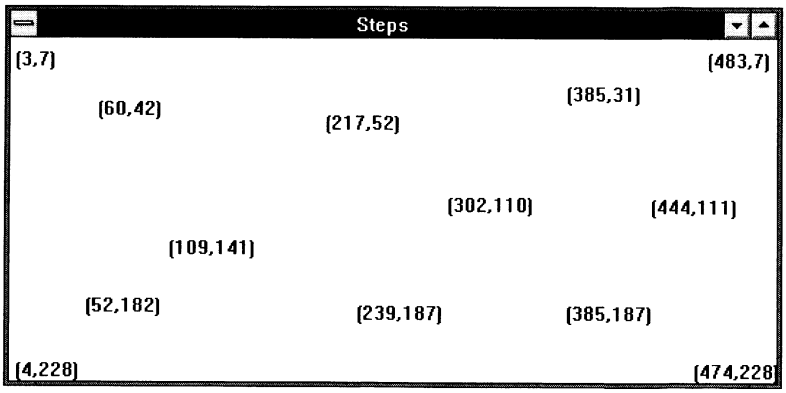
If you've worked with graphics before, you're familiar with the notion of coordinate systems. Just remember that in Windows, you also need to consider coordinates for your text.

The client area is the part of the window inside the frame.

When drawing, you only need to be concerned with the coordinates within your display context. Windows makes sure the display context falls within the *client area* of the window.

In this step, *Steps* draws text showing the coordinates of the point where you click your mouse in the window. For example, '(20, 30)' is the point 20 pixels right of the upper-left corner of the window's drawing surface and 30 pixels down. You draw right at the clicked point. Figure 2.1 shows *Steps* through Step 2.

Figure 2.1
Drawing text where the user clicks



Message parameters

A left button click event generates a *wm_LButtonDown* message, which you intercept with a message-response method called *WMLButtonDown*.

The *Msg* parameter of a message-response method carries valuable information about the event that produced the message, such as the point on which the user clicked. *Msg* is a *TMessage* record with fields to hold the Longint parameter, *lParam*, and the Word parameter, *wParam*. The identifiers *lParam* and *wParam* are the same as the corresponding fields in Windows' own message structure, *TMsg*.

TMessage also defines variant fields to hold subfields of *lParam* and *wParam*. For example, *Msg.lParamLo* holds the low-order word of *lParam*, while *Msg.lParamHi* holds the high-order word. Most often, you use the fields *wParam*, *lParamLo*, and *lParamHi*.

The meaning of the parameters depends on the message. See the online Help for details on each message and its parameters.

In the case of *WMLButtonDown*, *Msg.lParamLo* holds the x-coordinate of the clicked point, while *Msg.lParamHi* holds the y-coordinate. Thus, to rewrite *WMLButtonDown* to draw text showing the coordinates of the clicked point, you need to convert *Msg.lParamLo* and *Msg.lParamHi* into strings and concatenate them with two parentheses and a comma to make strings like '(25,34)'. This example uses the Windows function *WVSPrintF* to format the string.

Windows expects strings to be null-terminated (ending with a zero byte). See Chapter 18 in the Language Guide for details on using these strings.

Once you obtain the final string, you can draw it at the point that was clicked by calling the Windows function *TextOut*. Obtain the display context before drawing and release it afterward.

```
procedure TStepWindow.WMLButtonDown(var Msg: TMessage);
```

```
var S: array[0..9] of Char;
begin
  WVSPrintF(S, '(%d,%d)', Msg.LParam);
  DragDC := GetDC(HWindow);
  TextOut(DragDC, Msg.LParamLo, Msg.LParamHi, S, StrLen(S));
  ReleaseDC(HWindow, DragDC);
end;
```

Clearing the window

One more function you can add to the text drawing application is clearing the window. Notice that once you resize the window or cover and reveal it, the drawn text is erased. However, you might want to force screen clearing in response to a menu choice or some other user action, such as a mouse click.

To clear the window in response to a right mouse-button click, redefine the *WMRButtonDown* method to call the Windows procedure *InvalidateRect*, which causes the whole window to be repainted. Since your window doesn't yet know how to repaint itself, it just clears its client area:

```
procedure TStepWindow.WMRButtonDown(var Msg: TMessage);
begin
  InvalidateRect(HWindow, nil, True);
end;
```

You can see the current source code in the file STEP02.PAS.

Step 3: Drawing lines in the window

```
Step 1:  Basic App
Step 2:  Text
Step 3:  Lines
Step 4:  Menu
Step 5:  About Box
Step 6:  Pens
Step 7:  Painting
Step 8:  Streams
Step 9:  Printing
Step 10: Palette
Step 11: BWCC
Step 12: Custom ctls
```

Now that you've seen the drawing model (obtain a display context, draw, release the display context), you can use it in a more complete, interactive graphics application. In the next few steps, you'll build a simple painting program that lets the user draw on the main window.

Step 3 adds the following behavior:

- Left mouse-button clicks and drags connect dots along the way, resulting in drawn lines.
- Right mouse-button clicks bring up an input dialog box, allowing the user to change the line width.

To complete these steps, first study the Windows dragging model, then implement a simple graphics drawing program.

Dragging the line

You have already seen that a left mouse-button click results in a *wm_LButtonDown* message and a *WMLButtonDown* method call. In Step 1 (page 8), your program responded to left mouse-button clicks by bringing up message boxes. In Step 2 (page 19), it responded by drawing text on the screen. You also saw that a right mouse-button click results in a *wm_RButtonDown* message and a *WMRButtonDown* method call. You responded to right mouse-button clicks by clearing the window.

But these responses only cover the initial clicks of a mouse button. Many interactive Windows programs require the user to click and drag the mouse around on the screen to draw lines or rectangles or to place graphics in particular locations. For your graphics drawing program, you want to capture the dragging events and respond by drawing lines.

wm_MouseMove messages

You do this by responding to a few more messages. Windows sends *wm_MouseMove* when the user drags the mouse to a new point in a window and *wm_LButtonUp* when the user releases the left mouse button. Typically, a window receives one *wm_LButtonDown* message, followed by a series of *wm_MouseMove* messages (one for each point dragged over), followed by one *wm_LButtonUp* message.

A typical graphical Windows program responds to *wm_LButtonDown* by initiating the drawing process (obtaining a display context, among other things). It responds to *wm_MouseMove* by drawing or moving graphics, and it responds to *wm_LButtonUp* by ending the drawing process (releasing the display context).

Responding to drag messages

Remember that *wm_LButtonDown* is always followed by *wm_LButtonUp*, with or without *wm_MouseMove* messages in between. Therefore, every time you obtain a display context, you later release it. In this case, you obtain a single display context for all the drawing that takes place between the user clicking the left mouse button and releasing it.

Releasing every display context you obtain is critical to the proper functioning of any Windows program. However, you can also

add one more safety measure. Define a new Boolean field in *TStepWindow*, the main window type, called *ButtonDown* and make sure to initialize it to *False* in *TStepWindow.Init*. *WMLButtonDown* sets it to *True* and *WMLButtonUp* sets it back to *False*. Then you can check the value of *ButtonDown* before obtaining and releasing the display context.

Here are the three mouse drag methods:

```
procedure TStepWindow.WMLButtonDown(var Msg: TMessage);
begin
  InvalidateRect(HWindow, nil, True);
  if not ButtonDown then
  begin
    ButtonDown := True;
    SetCapture(HWindow);
    DragDC := GetDC(HWindow);
    MoveTo(DragDC, Msg.lParamLo, Msg.lParamHi);
  end;
end;

procedure TStepWindow.WMMouseMove(var Msg: TMessage);
begin
  if ButtonDown then
    LineTo(DragDC, Msg.lParamLo, Msg.lParamHi);
end;

procedure TStepWindow.WMLButtonUp(var Msg: TMessage);
begin
  if ButtonDown then
  begin
    ButtonDown := False;
    ReleaseCapture;
    ReleaseDC(HWindow, DragDC);
  end;
end;
```

Drawing points and lines

MoveTo and *LineTo* are graphics functions in the Windows API that move the current drawing position and draw a line to the current position, respectively. They require a handle to the display context, *DragDC*, to function properly. Remember that you're not drawing directly on the window, but on its display context.

Capturing the mouse

SetCapture and *ReleaseCapture* are Windows functions that ensure the proper sending of the *wm_MouseMove* messages by Windows. For example, if you drag the mouse outside of the window, Windows will still send *wm_MouseMove* messages to the main window, rather than to an adjacent window you might happen to pass over. Capturing the mouse also ensures that the mouse-up message goes to your window, so it knows to stop drawing, even if the mouse has moved over another window.

Be sure to update the object definition for *TStepWindow* with method headers for *WMMouseMove* and *WMLButtonUp*:

This brings the code up to STEP03A.PAS.

```
procedure WMLButtonUp(var Msg: TMessage); virtual wm_First +
  wm_LButtonUp;
procedure WMMouseMove(var Msg: TMessage); virtual wm_First +
  wm_MouseMove;
```

Changing the pen size

At this point, you can only draw thin black lines. But drawing programs traditionally let you change the thickness of the lines you draw. When you do this, you're not really changing the thickness of the *lines*, but the size of the pen you use to draw the lines. Pens, as well as brushes, fonts, and palettes, are the drawing tools embodied by a display context. In this step, you'll learn how to set new tools in a display context while giving *Steps* the ability to set a new line thickness.

You'll also use an input dialog box (of type *TInputDialog*) to provide a mechanism for the user to change the size of the pen. Figure 2.2 shows the input dialog in use.

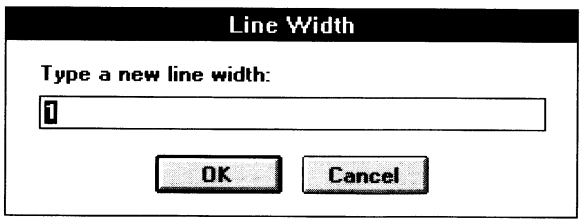


Figure 2.2
Getting a new line width with an input dialog box

Tracking pen size

In order to change the thickness of the drawn lines, you need to first understand a little more about Windows graphics, and display contexts in particular.

Drawing tools

Windows uses several *drawing tools* to produce graphics and text in a window: pens, brushes, and fonts. These drawing tools are window elements stored in Windows memory, not unlike visible screen elements like windows and controls. Your program accesses drawing tools with handles, just as it does with windows. Since ObjectWindows does not use objects to represent drawing tools, your programs need to create them and then delete them from Windows memory when you are done with them.

In Step 6 (p. 43), you'll create an object to encapsulate one drawing tool, the pen.

Think of a drawing tool as a painter's paintbrush and a display context as the canvas. The painter first creates drawing tools (paintbrushes) and obtains a display context (canvas). The painter then selects the proper drawing tools, using only one brush at a time. Similarly, a Windows program must select drawing tools into a display context.

Default drawing tools

So, how is it that you could draw text and lines in your windows without selecting any drawing tools? All display contexts come with a set of default tools: a thin black pen, a solid black brush, and a system font. In this step, you select a different, thicker pen for drawing in the window.

Getting a new pen size

The first step is to provide a way to choose a new pen width. An easy way to do this is to use the input dialog box from the *OStdDlgs* unit. Add *OStdDlgs* to the **uses** statement. To use the Windows-compatible string manipulation functions, also use *Strings*. The start of your program file should now look like this:

```
program Steps;

uses Strings, WinTypes, WinProcs, OWindows, OStdDlgs;
  ⋮
```

Running the input dialog box

An input dialog is a simple dialog box that prompts for and returns one line of text input. You can use it without modifying *TInputDialog* or any of its methods.

Clicking the right mouse button is a convenient way to bring up the option to change the pen width. Let's redefine the *WMRButtonDown* method to bring up an input dialog box.

Since the input dialog box appears for only a short time and all of its processing can be handled by one method, you need not define it as a field of *TStepWindow*. It can exist as a local variable of the *WMRButtonDown* method. You construct and dispose of the input dialog box object, all within the *WMRButtonDown* method.

Executing dialog boxes is explained in detail in Chapter 11, "Dialog box objects."

Once *Init* constructs the input dialog box object, you can run it as a modal dialog box by calling *ExecDialog*. *ExecDialog* checks to make sure *Init* succeeded, creates the dialog box object's corresponding screen element, and executes the dialog box. Processing for *ExecDialog* ends only after the user has closed the dialog by clicking OK or Cancel.

If the user clicks OK, *InputText* is filled with the user's input by calling the *GetText* method of *TInputDialog*. Since you are asking for a thickness number, you must convert the returned text to a number and pass it in a call to *SetPenSize*. Thus, every time the user chooses a new line thickness, delete the old pen and create a new pen.

```
procedure TStepWindow.WMRButtonDown(var Msg: TMessage);
var
  InputText: array[0..5] of Char;
  NewSize, ErrorPos: Integer;
begin
  if not ButtonDown then
  begin
    Str(PenSize, InputText);
    if Application^.ExecDialog(New(PInputDialog,
      Init(@Self, 'Line Thickness', 'Input a new thickness:',
        InputText, SizeOf(InputText)))) = id_Ok then
    begin
      Val(InputText, NewSize, ErrorPos);
      if ErrorPos = 0 then SetPenSize(NewSize);
    end;
  end;
end;
```

Adding object fields

Next, add a field to *TStepWindow* to store a handle to the pen tool you'll use to draw graphics. In this program, you limit yourself to drawing and displaying lines in only one thickness at a time. The pen corresponding to this thickness is stored in the new *TStepWindow* field called *ThePen*. You also write a method, *SetPenSize*, to create the new pen tool and delete the old pen tool. Your *TStepWindow* object declaration should now look like this:

```
type
  PStepWindow = ^TStepWindow;
  TStepWindow = object(TWindow)
    DragDC: HDC;
    ButtonDown, HasChanged: Boolean;
    ThePen: HPen;
    PenSize: Integer;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    destructor Done; virtual;
    function CanClose: Boolean; virtual;
    procedure WMLButtonDown(var Msg: TMessage); virtual wm_First +
      wm_LButtonDown;
    procedure WMLButtonUp(var Msg: TMessage); virtual wm_First +
      wm_LButtonUp;
    procedure WMMouseMove(var Msg: TMessage); virtual wm_First +
      wm_MouseMove;
    procedure WMRButtonDown(var Msg: TMessage); virtual wm_First +
      wm_RButtonDown;
    procedure SetPenSize(NewSize: Integer); virtual;
  end;
```

Initializing the fields

To initialize these new fields, you need to modify the *Init* constructor to set up the pen and override the *Done* destructor to dispose of the pen. Remember to call the inherited methods in the new methods:

```
constructor TStepWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  ButtonDown := False;
  HasChanged := False;
  PenSize := 1;
  ThePen := CreatePen(ps_Solid, PenSize, 0);
end;

destructor TStepWindow.Done;
begin
  DeleteObject(ThePen);
  inherited Done;
end;
```

Drawing the lines

Now, alter the *WMLButtonDown* method to select the current pen (*ThePen*) into the newly obtained display context. Like *MoveTo* and *MessageBox*, *SelectObject* is a Windows API function.

```
procedure TStepWindow.WMLButtonDown(var Msg: TMessage);
begin
  if not ButtonDown then
  begin
    ButtonDown := True;
    SetCapture(HWindow);
    DragDC := GetDC(HWindow);
    SelectObject(DragDC, ThePen);
    MoveTo(DragDC, Msg.lParamLo, Msg.lParamHi);
  end;
end;
```

The above method selects the already-created pen into the display context. However, to create the pen, you need to write the following *SetPenSize* method, which *WMRButtonDown* calls:

This makes STEP03B.PAS.

```
procedure TStepWindow.SetPenSize(NewSize: Integer);
begin
  DeleteObject(ThePen);
  ThePen := CreatePen(ps_Solid, NewSize, 0);
  PenSize := NewSize;
end;
```

Calling the Windows function *CreatePen* is one way to create a Windows pen tool with the specified thickness. You store a handle to the pen in *ThePen*. Deleting the previous pen is a very important step. Without this step, you would slowly use up Windows memory with no way of recovering it.

In steps 5 and 6, you'll create your own dialog box and a pen object and use them to create fancier drawings.

C H A P T E R

3

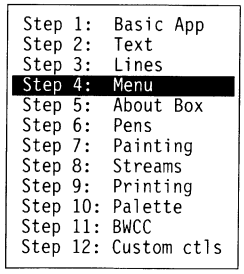
Menu and dialog resources

Most Windows applications have menus in their main windows to provide selections for the user, such as File | Save, File | Open, and Help. In Step 4, you'll add a menu bar to *Steps*. User input and option selection in Windows programs often take place in dialog boxes. Step 5 will add a dialog box to *Steps*.

What these menus and dialog boxes have in common is their use of *resources*. Resources are data stored in an executable file that describe user interface elements for Windows applications. The resource doesn't contain an actual menu or dialog box, but rather it contains the information a program must have to create a particular menu or dialog box.

The easiest way to create resources for your programs is to use visual resource editors, such as those in Borland's Resource Workshop. For full information on creating and editing resources, see the *Resource Workshop User's Guide*.

Step 4: Adding a menu bar



In a windowing environment, a menu selection is in the same category as a mouse click: they're both user events. Responding to a menu selection is a lot like responding to other user events. This section traces the required steps to add a menu to an application:

- Design the menu as a menu resource
- Define menu constants in an include file
- Specify the resource file from the program
- Load the menu resource into the main window object
- Define responses to menu selections

Window attributes are explained in detail in Chapter 10, "Window objects."

An application's menu is not a separate object, but an *attribute* of the main window. All window objects have a set of attributes stored in a record in the object's *Attr* field. The menu, or rather a handle to the menu, is stored in the *Menu* field of *Attr*. To set the menu attribute, you redefine the constructor for your window type, *TStepWindow*.

Menu resources

The definition of the menu items is not part of the program's source code. Instead, there is a resource that contains the text of the menu selections and the structure of the top-level items and their subitems. Use Resource Workshop to design your menus and other resources, such as dialog boxes, icons, and bitmaps.

Defining resource IDs

An application accesses its attached resources by specifying the resource ID. This ID is an integer, such as 100, or an integer constant, such as *MyMenu*. In addition, an application distinguishes one menu selection from another by the menu ID associated with each menu item.

Defining menu constants

To make the code more readable, substitute the menu IDs with constants you define in an include file. When you create your menu resources using Resource Workshop or the resource compiler, you can include these same constants and use the same identifiers to create the resource that you use when you access it in your program. *Steps* defines its menu ID constants in the file STEPS.INC:

These constants are defined in STEPS.INC.

```
const
  cm_FilePrint = 105;
  cm_FileSetup = 107;
  cm_Pen       = 200;
  cm_About     = 201;
  cm_PalShow   = 301;
  cm_PalHide   = 302;
```

Note that a number of menu item IDs are not defined in STEPS.INC. That's because ObjectWindows defines a number of common menu command constants for you, including *cm_FileOpen*, *cm_FileNew*, *cm_FileSave*, and *cm_FileSaveAs* in the file OWINDOWS.INC.

Including resource files

To continue with *Steps*, use Resource Workshop or the resource compiler to create a menu resource and save it as a .RES file, STEPS.RES. See the file STEPS.RC for the resource file in source format. You can also use the STEPS.RES file provided on the distribution disks. Once you have STEPS.RES, include it by using the **$R** compiler directive like this:

```
{$R STEPS.RES}
```

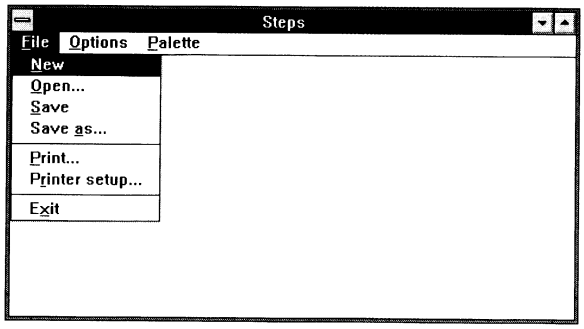
For information on modifying resources already linked to executable files, see the Resource Workshop User's Guide.

The **$R** compiler directive automatically appends the specified resource file to the executable file at the end of the compile and link. Resources can also be added to or removed from existing executable files, and existing resources can be modified.

Don't confuse the menu's resource ID with the menu IDs of the individual menu items.

Figure 3.1 shows the appearance of this menu (resource ID 100). It includes File, Options, and Palette selections, and the File menu has the items New, Open, Save, Save As, Print, Printer Setup, and Exit. Top-level menus that have subitems, such as File, do not have menu IDs, and selecting them causes no action besides displaying their subitems. If a top-level menu item has no subitems, it has a menu ID and can cause some action.

Figure 3.1
Steps with a menu resource



Loading the menu resource

Obtain the menu resource by calling the *LoadMenu* Windows function:

```
LoadMenu(HInstance, MakeIntResource(100));
```

*To use the PChar type, the **$X+** compiler directive must be on (the default setting).*

MakeIntResource(100) casts the number 100 into a pointer type called *PChar*, which is a pointer to an array of characters. Windows functions that receive strings as arguments require them to be of the *PChar* type. When dealing with resources, Windows expects integer numbers to be represented in *PChar* form, so if you want to access a resource that has a numeric ID, you need to typecast it with *MakeIntResource*.

As an alternative, a menu resource might have a symbolic identifier, such as `'SAMPLE_MENU'`. In that case, load the menu resource as follows:

```
LoadMenu(HInstance, 'SAMPLE_MENU');
```

Here's how *TStepWindow.Init* should now look. Note that the first thing it does is call the *Init* constructor it inherits from *TWindow* to perform the initialization required of all window objects:

```
constructor TStepWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, MakeIntResource(100));
  ButtonDown := False;
  HasChanged := False;
end;
```

Now when the main window is displayed, it has the operational menu shown in Figure 3.1. In order to make the menu selections do something, however, you must intercept and respond to the

menu messages. If you haven't defined a response to a menu command, you can choose the menu item, but nothing happens.

Intercepting the menu message

When the user chooses a menu item, the window to which the menu is attached receives a Windows *command message*. ObjectWindows handles and dispatches these *wm_Command* messages in much the same way as other messages, but makes it easier for you to handle specific commands.

One of the parameters of a *wm_Command* message is the command itself (a number corresponding to the menu ID of the item chosen). Instead of calling a *WMCommand* method and making you decide what to do with each possible command, ObjectWindows calls methods based on the specific commands. To process these messages, you define methods for the *TStepWindow* object type using a special extension:

```
procedure CMFileNew(var Msg: TMessage);
  virtual cm_First + cm_FileNew;
```

Message ranges and offsets are explained more thoroughly in Chapter 16.

where *cm_First* is an ObjectWindows-defined constant defining the beginning of the range of constants for commands and *cm_FileNew* is the desired menu ID. This means that all menu items need to have unique IDs, unless you intend to have them invoke the same response.

Don't confuse this dynamic method index based on *cm_First* with the one for responding to incoming Windows messages (based on the offset *wm_First*). *cm_First* is a special offset used only to define response methods for menu and accelerator commands.

Defining command-response methods

Now, you can define all of the command-response methods:

```
procedure CMFileNew(var Msg: TMessage);
  virtual cm_First + cm_FileNew;
procedure CMFileOpen(var Msg: TMessage);
  virtual cm_First + cm_FileOpen;
procedure CMFileSave(var Msg: TMessage);
  virtual cm_First + cm_FileSave;
procedure CMFileSaveAs(var Msg: TMessage);
  virtual cm_First + cm_FileSaveAs;
procedure CMFilePrint(var Msg: TMessage);
  virtual cm_First + cm_FilePrint;
procedure CMFileSetup(var Msg: TMessage);
  virtual cm_First + cm_FileSetup;
```

You don't have to define a *CMExit* procedure, because *TWindowsObject* already defines one that terminates the program if a main window sees a *cm_Exit* command.

Linking keys to commands

Command-response methods are not linked to any particular menu item; they are keyed to a particular command. The object doesn't care where the command came from, it just knows that something caused it to get this command message. You can therefore have more than one way to generate a command. A common way to do this is to use *accelerators*, sometimes called shortcuts.

Accelerators are defined in resources, just like menus, but they are much simpler. An accelerator resource is just a table of keystrokes and the commands they generate. For information on creating accelerator resources, see the *Resource Workshop User's Guide.*

Each application can have only one set of accelerators. To load an accelerator resource into your application, override the application object's *InitInstance* method:

```
procedure TMyApplication.InitInstance;
begin
  inherited InitInstance;
  HAccTable := LoadAccelerators(HInstance, 'ShortCuts');
end;
```

The 'ShortCuts' accelerators in STEPS.RES bind familiar function keys from the IDE to similar functions in *Steps*. For example, *F3* generates the command *cm_FileOpen*.

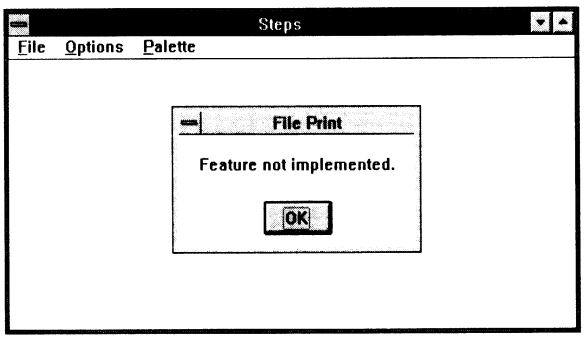
Responding to menu commands

For each menu selection, you now have a method that will be invoked. For example, choosing File | Print invokes your *CMFilePrint* method. For now, just display a message box:

```
procedure TStepWindow.CMFilePrint(var Msg: TMessage);
begin
  MessageBox(HWindow, 'Feature not implemented.',
    'File Print', mb_OK);
end;
```

Figure 3.2 shows *Steps'* response to the File | Print selection.

Figure 3.2
Steps responding to File | Print



For *CMFileOpen*, *CMFileSave*, *CMFileSaveAs*, and *CMFileSetup*, write dummy methods similar to *CMFilePrint*. Later, you'll rewrite these methods to perform meaningful actions.

At this point, you can respond to the File | New menu selection in a more interesting way, by clearing the window. Add the following *CMFileNew* method:

```
procedure TStepWindow.CMFileNew(var Msg: TMessage);
begin
  InvalidateRect(HWindow, nil, True);
end;
```

InvalidateRect forces a repainting of the window. Since there are no lines to redraw, the window becomes blank. The complete source code to *Steps* up to this point is in the file STEP04A.PAS.

Adding a stock dialog box

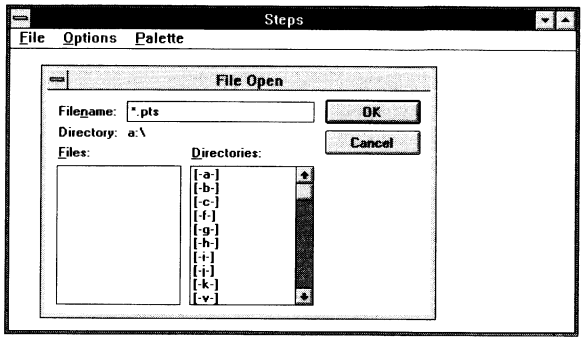
A dialog box is like a pop-up window, but it usually stays on the screen for a short period of time and performs one particular input-related task, such as choosing a printer or setting up a document page. Here, you add a dialog box to *Steps* for opening and saving files.

The file dialog box, one of ObjectWindows' stock dialog boxes, is defined by the type *TFileDialog*. The file dialog box is useful in any situation where you ask the user to pick a disk file for saving or loading. For example, a word processing application would use a file dialog box for opening and saving documents.

You will bring up a file dialog box in response to the user's selection of File | Open or File | Save As. The file dialog box replaces the "Feature not implemented" message box. In Step 8 (page 59), you'll hook it up to some real files and save and open

them to store and retrieve real data. For now, just show the dialog boxes. Figure 3.3 shows the appearance of the file dialog box.

Figure 3.3
Steps with the File Open dialog box



Adding the file dialog box to *Steps* takes three steps:

- Add an object field to hold the file name
- Modify the window object's constructor to initialize the file name
- Run the dialog box

Adding an object field

Instead of storing an entire file dialog box object as a field of its parent window, you should construct a new file dialog box object each time you need one. What you store instead is just the data you want to use with the file dialog box: a file name and a file mask. This is good practice: Instead of keeping entire objects around when you may never need them, simply store the data you would need to initialize the objects, should you need them.

Constructing a file dialog box takes three parameters: a parent window, a resource template, and a file name or mask, depending on whether the dialog is for opening or closing a file. The resource template specifies which of the standard file dialog boxes you want to use. The standard file dialog resources are identified by the resource IDs *sd_FileOpen* and *sd_FileSave*. The file-name parameter is used to pass a default file mask to the file open dialog (as well as returning the selected file name) and to pass the default name for file saving.

The resource template parameter determines whether the file dialog box will be used to open a file or save a file. If the dialog resource has a file list box with the control ID *id_FList*, the dialog box is for opening files; the lack of such a list box indicates the dialog box is for saving files.

The *TStepWindow* type definition should now look like this:

You need to use the WinDos unit to use the fsPathName constant.

```
TStepWindow = object(TWindow)
  :
  FileName: array[0..fsPathName] of Char;
  :
```

Modifying the constructor

You created an *Init* constructor for type *TStepWindow* to instantiate the help window object. Now you need to add to it the code to initialize *FileName*:

```
StrCopy(FileName, '*.PTS');
```

The .PTS extension is used on files holding the points of your drawings.

Running the dialog box

Depending on the resource template parameter passed to the constructor of a file dialog object, the dialog box can support either file opening or saving. Either option produces a dialog box similar to the one shown in Figure 3.3. There are two differences between the file open and the file save dialog: The file open dialog has a list of files in the current directory matching the current file mask, while the file save dialog initially shows the current file name in the edit area of the dialog's edit control, but has no list of files.

Here's how to rewrite *CMFileOpen* and *CMFileSaveAs*:

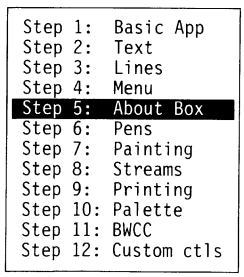
```
procedure TStepWindow.CMFileOpen(var Msg: TMessage);
begin
  if Application^.ExecDialog(New(PFileDialog,
    Init(@Self, PChar(sd_FileOpen), FileName))) = id_Ok then
    MessageBox(HWindow, FileName, 'Open the file:', mb_Ok);
end;

procedure TStepWindow.CMFileSaveAs(var Msg: TMessage);
begin
  if Application^.ExecDialog(New(PFileDialog,
    Init(@Self, PChar(sd_FileSave), FileName))) = id_Ok then
    MessageBox(HWindow, FileName, 'Save the file:', mb_Ok);
end;
```

Notice that running the file dialog box uses the same *ExecDialog* method you called to run the input dialog box in Step 3 (page 23). All modal dialog boxes are executed with the application object's *ExecDialog* method.

The full source code to *Steps* to this point is in the file STEP04B.PAS.

Step 5: Adding a dialog box



Thus far, *Steps* uses two very simple dialog boxes: the message box in the *CanClose* method and the input dialog box for changing the size of the pen. These stock dialog boxes are convenient for simple tasks, but your programs usually require more complex, application-specific interactions with the user. For this, you can design your own dialog boxes.

Like a menu, a dialog box is usually created from a description stored in a resource. For complex dialog boxes, this is much faster than creating each element of the whole window individually. However, unlike a menu, ObjectWindows uses an object to represent the dialog box, because programs have to interact with dialog boxes in more varied and complicated ways.

Creating a dialog box from a resource requires the following steps:

- Creating a dialog box resource
- Constructing a dialog box object
- Executing the dialog box

Creating dialog box resources

Remember, a resource is just a description of something your program will create.

You can design dialog box resources in several ways, using Resource Workshop or the resource compiler. A dialog box is a specialized frame window that contains one or more controls, such as buttons, list boxes, and icons. Your application doesn't know or care what the controls look like or where they are positioned; it only knows what kind of controls they are and what IDs they have.

Control IDs

Like the items in a menu resource, each control in a dialog box resource has an ID that applications use to identify which control they want to interact with. Controls that have no interaction with the program, such as static text or bitmaps, don't need unique IDs and usually have an ID of –1.

Normally, if your application needs to access a particular control, you assign a constant identifier to the control's ID, so both your program and the resource compiler can use the same symbolic name for the ID. You should define such ID constants in an include file or in a unit that contains only constant declarations. Resource Workshop manages such files for you when you edit your resources.

Constructing a dialog box object

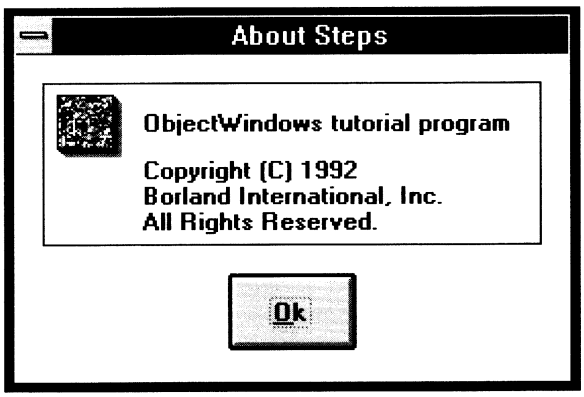
Once you have a dialog box resource defined, your program can use it to create and execute a dialog box. Dialog boxes usually show up as child windows of the application's main window, but they are created somewhat differently than regular windows.

The constructor for a dialog box object looks just like that for a window object, taking two parameters. In both cases, the first parameter is a pointer to the parent-window object. The second parameter, a *PChar*, defines the caption of a window object. For a dialog box object, however, it is the name of the dialog resource to use as a template for the dialog box.

The resource file for *Steps* defines a dialog box called 'ABOUTBOX' that you can use as an About box, as shown in Figure 3.4. Constructing a dialog box object from this resource looks like this:

```
New(PDialog, Init(@Self, 'ABOUTBOX'));
```

Figure 3.4
The About box for Steps



Executing the dialog box

To execute a custom dialog box, use the same *ExecDialog* method you've already used for stock dialog boxes:

```
Application^.ExecDialog(New(PDialog, Init(@Self, 'ABOUTBOX')));
```

Of course, you also need to define a command to bring up the About box; *Steps* uses *cm_About*, generated by the Options | About menu choice. By now, this sort of command-response method should look quite familiar:

This makes STEP05.PAS.

```
type
  TStepWindow = object(TWindow)
      :
    procedure CMAbout(var Msg: TMessage);
      virtual cm_First + cm_About;
  end;

procedure TStepWindow.CMAbout(var Msg: TMessage);
begin
  Application^.ExecDialog(New(PDialog, Init(@Self, 'ABOUTBOX')));
end;
```

In Step 6 (page 43), you'll create and use a more complicated dialog box with multiple controls you can manipulate.

Modal and modeless dialog boxes

Once you execute a dialog box with *ExecDialog*, the dialog box becomes *modal*, meaning that is the mode the program is in until the dialog box is closed. Nothing else in the program receives messages while the modal dialog box is active. Such a dialog box is called *application-modal*, because it is only modal with respect to the application that executed it. There are also *system-modal* dialog boxes that freeze all activity in all applications until the dialog box closes. These are very rare and should only be used when having other applications running could cause problems.

Sometimes you want to have a dialog box that stays around while you use other parts of your program. Such a dialog box acts almost like a regular window, but it's not modal, so it's called *modeless*. For information on creating a modeless dialog box, see Chapter 11, "Dialog box objects."

C H A P T E R

4

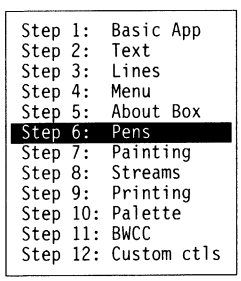
Working with a dialog box

Now that you've made *Steps* pop up a dialog box created from a resource, you need to learn how to communicate with a dialog box. First you'll create an object that encapsulates all the properties of Windows' pen tool, then you'll use a dialog box to set and change those properties.

This chapter includes the following steps:

- Defining a pen object
- Creating a complex dialog box
- Adding control objects
- Creating a transfer buffer
- Executing the dialog box
- Reading the results

Step 6: Changing pen attributes



This step covers a number of topics involving Windows drawing tools, specifically the pen tool used for drawing lines. Windows pens have three separate attributes: a style, a width, and a color.

The first part of this step, creating an object to represent the pen, is not absolutely necessary, but it allows your window to deal with the pen as a single entity, rather than having to keep track of all the pen's attributes separately. By encapsulating the pen, you can also avoid dealing with some of the repetitious details of

using GDI tools, much as ObjectWindows' window objects shield you from some of the details of creating windows.

Creating a pen object

Although Windows refers to its drawing tools as "objects" (hence names like *SelectObject* and *DeleteObject*), they are not objects in the true object-oriented sense, since they don't inherit and they aren't polymorphic. A pen is actually just a group of three drawing characteristics referred to by Windows when it draws a line. In reality, these characteristics are just properties of the display context, but it is useful to think of them as being embodied in a pen.

The properties of a pen

Most of the code in this chapter comes from PEN.PAS. STEP06A.PAS and STEP06B.PAS have minor changes to use the Pen unit.

TPen is defined in the Pen unit.

The three drawing properties that make up a pen are its style, size, and color. In Step 3 (page 23), you changed the size of the pen and kept track of the current size of the pen in a field in the window object. Instead of making three separate fields to track each of the pen's properties, you can encapsulate them all into a single object, a *TPen*. The declaration of *TPen* is shown here:

```
type
  PPen = ^TPen;
  TPen = object(TObject)
    Width, Style: Integer;
    Color: Longint;
    constructor Init(AStyle, AWidth:Integer; AColor: Longint);
    constructor Load(var S: TStream);
    procedure ChangePen;
    procedure Delete;
    procedure Select(ADC: HDC);
    procedure SetAttributes(AStyle, AWidth: Integer;
      AColor: Longint);
    procedure Store(var S: TStream);
  private
    PenHandle, OldPen: HPen;
    TheDC: HDC;
    PenData: TPenData;
  end;
```

The *Init* constructor creates a new pen object with the given style, size, and color. *SetAttributes* changes the attributes of an already-created pen object. *ChangePen* brings up the dialog box that enables the user to specify pen attributes. *Load* and *Store* allow pen objects to be stored on a stream.

Selecting and deleting pen objects

Select and *Delete* do the most interesting work. *Select* creates a Windows drawing tool based on the properties stored in the attribute fields. Instead of having the drawing program call the Windows API to create a pen, get its handle, select the pen into the display context, use the pen, then delete the pen, you construct a pen object, then when you're ready to use it, you select it, use it, and (maybe) delete it.

Delete disposes of the pen handle, freeing up that resource to Windows. *Select* checks to see if there is already a pen selected and disposes of an existing pen before creating and selecting a new one. This is useful if the same pen is going to be reused continuously, so you don't have to call *Delete* every time you use a pen. On the other hand, in Step 7 (page 53), you'll store the lines you draw, and each line will have its own pen object. If each pen object created and held onto a Windows pen, Windows would soon run out of resources. In that instance, it's important to call a pen object's *Delete* method right after using the pen.

The nicest part of using *TPen* is that you no longer have to bother with getting, storing, and deleting the pen handle. *TPen* has two private fields, one of which stores the pen handle. The pen object keeps track of the handle and the interactions with Windows; your program just deals with the object. The other private field, *PenData*, holds a transfer buffer used in this step.

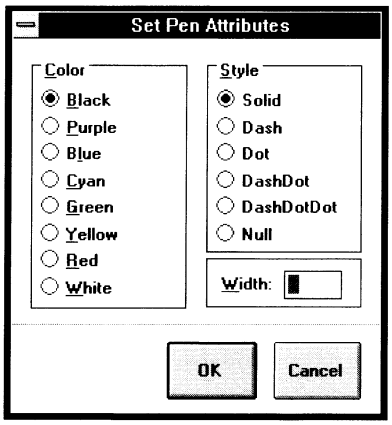
STEP06A.PAS contains the code for *Steps*, modified to use the *TPen* object in the *Pen* unit. Most of the changes are small, such as changing the field *ThePen* from type *HPen* to type *PPen* and replacing the *SetPenSize* method with a call to the pen object's *SetPenAttributes* method, since the pen object can control color and style as well as size.

Creating a complex dialog box

The dialog box used as an About box in Step 5 (page 40) is quite simple. The only control a user has to interact with is the OK button, and that is managed by *TDialog*'s default behavior. The real power of dialog box objects comes into play when you have to set and read the values of the controls in the dialog box.

The *Pen* unit defines a more complicated dialog box resource called 'PenDlg' that gives options for changing the attributes of the pen object you just defined. Figure 4.1 shows this dialog box.

Figure 4.1
The dialog box for changing pen attributes



Constructing an object from the 'PenDlg' resource works exactly as it did for the About box with the exception of the parent window. Because the pen attribute dialog box is executed from within the *TPen* object rather than inside a window object, you can't use *@Self* as the parent window. Instead, *TPen* attaches the dialog box to the one window it knows will *always* be there, the application's main window:

```
procedure TPen.ChangePen;
var PenDlg: PPenDialog;
begin
  :
  PenDlg := New(PPenDialog, Init(Application^.MainWindow,
    'PenDlg'));
  :
end;
```

The other main difference is that this time you've derived a new object type, *TPenDialog*. Since the About box didn't use anything other than the default dialog box behavior encapsulated in *TDialog*, you didn't have to create a new object type for it. The pen attribute dialog, however, must behave in certain unique ways, so you must customize its object.

Listing 4.1 shows the definition of *TPenDialog* from the *Pen* unit:

Listing 4.1
TPenDialog's definition

```
type
  PPenDialog = ^TPenDialog;
  TPenDialog = object(TDialog)
    constructor Init(AParent: PWindowsObject; AName: PChar);
  end;
```

```
constructor TPenDialog.Init(AParent: PWindowsObject; AName: PChar);
var
  AControl: PRadioButton;
  i: Integer;
begin
  inherited Init(AParent, AName);
  AControl := New(PEdit, InitResource(@Self, 1099, 7));
  for i := 0 to 7 do
    AControl := New(PRadioButton, InitResource(@Self, 1100 + i));
  for i := 0 to 5 do
    AControl := New(PRadioButton, InitResource(@Self, 1200 + i));
end;
```

The control objects constructed in *TPenDialog.Init* are explained in the next section.

Associating control objects

If your program needs to interact directly with controls in a dialog box (for example, to put items into a list box or to determine whether a check box has been checked), you probably want to associate objects with those controls. Then you can manipulate the controls just like any other objects in your application.

Using interface objects

In "normal" Windows programming (that is, without ObjectWindows), your application has to communicate with every element on the screen through messages and Windows API functions. This is equally true for windows, dialog boxes, and controls. As you have already seen, ObjectWindows makes it easy to create and manipulate windows by insulating you from Windows as much as possible using objects to represent screen elements. These interface objects also make communicating with controls in dialog boxes much easier.

Interface objects are described in Chapter 9, and control objects, in particular, are described in Chapter 12.

If you do not use control objects, you can still interact with the controls, but it involves a great deal of interaction with the Windows API, sending messages to controls and interpreting the results. ObjectWindows makes this much easier by encapsulating the behavior of each control in an object. The same messages are passed and handled, but ObjectWindows takes care of the details for you.

Associating an object with a control created from a resource is simple: Inside the constructor of the dialog box object, you construct objects for any controls you want to manipulate.

However, instead of using *Init* to construct the control objects, you use *InitResource*.

The InitResource constructor

When you create a control object at run time (as opposed to creating it from a resource), you have to specify the location, size, and initial value or state of the control, as well as indicating what object is its parent. You pass all these items as parameters to the object's *Init* constructor.

Associating an object with a control from a resource is much simpler, since information like location and size are defined by the resource. All you have to give is the parent and the ID of the control to the *InitResource* constructor. Since control objects are usually constructed inside the constructor of their parent dialog boxes, the parent pointer is nearly always *@Self*.

The *Pen* unit's pen dialog associates objects with its edit control (for the pen size) and both sets of radio buttons (for the color and style), as shown in Listing 4.1.

Notice that all the control objects are constructed and assigned to the same local variable, *AControl*. Your program won't have to interact with any of these control objects directly, since the rest of the program is not active while the modal dialog box is running. *InitResource* adds the control objects to the dialog box object's child-window list to ensure that their screen elements are created and destroyed along with the dialog box.

In general, there is no reason to assign object fields to child windows in modal dialog boxes. In Step 11 (page 71), however, you'll see how storing pointers to controls in a modeless window object makes it easy to manipulate the controls.

Creating a transfer buffer

Now that you have objects associated with the controls in the dialog box, you need a way to set and read their values. This is done with a *transfer buffer*. A transfer buffer is a record that holds one field for each control in the dialog box being transferred to or from.

For example, the dialog box you created in Step 6 (page 43) has an edit field and fourteen radio buttons. The edit control needs a string passed to it, and each of the radio buttons gets a *Word* indicating whether it is checked. The *Pen* unit defines a record type for transfers to and from *TPenDialog*s:

```
type
  TPenData = record
    XWidth: array[0..6] of Char;
    ColorArray: array[0..7] of Word;
    StyleArray: array[0..5] of Word;
  end;
```

You could also handle the radio buttons as fourteen individual fields, or as a single array of fourteen *Words*; the data transfer would be the same. However, since your application will treat them as two groups of eight and six buttons, respectively, it is convenient to set up a field for each group.

Assigning the buffer

Every descendant of *TWindowsObject* has a *TransferBuffer* field. When you want to transfer data to a dialog box, you set the dialog box object's *TransferBuffer* to point to a transfer record:

```
PenDlg := New(PPenDialog, Init(Application^.MainWindow, 'PenDlg'));
PenDlg^.TransferBuffer := @PenData;
```

If your programs create dialog box objects dynamically, be sure they assign the transfer buffer each time. *TransferBuffer* is **nil** by default, which means no data get transferred.

Filling the buffer

Before you can actually transfer the data to the dialog box, you have to set the values of the fields in the transfer buffer. *TPen.ChangePen* does this before bringing up the pen dialog box:

```
procedure TPen.ChangePen;
var
  PenDlg: PPenDialog;
  TempWidth, ErrorPos: Integer;
begin
  SetColorAttr(PenData, Color);
  SetStyle(PenData, Style);
  wvsprintf(PenData.XWidth, '%d', Width);
  PenDlg := New(PPenDialog, Init(Application^.MainWindow,
    'PenDlg'));
  PenDlg^.TransferBuffer := @PenData;
  if Application^.ExecDialog(PenDlg) <> id_Cancel then
  begin
    Val(PenData.XWidth, TempWidth, ErrorPos);
    if ErrorPos = 0 then
      SetAttributes(GetStyle(PenData), TempWidth,
        GetColorAttr(PenData));
  end;
end;
```

SetColorAttr and *SetStyle* both take advantage of the fact that the transfer buffer sets up the radio buttons as arrays of *Word*s. *SetStyle*, for example, looks like this:

SetColorAttr does the same operation on ColorArray.

bf_Checked and bf_Unchecked are ObjectWindows constants.

```
procedure SetStyle(var ARec: TPenData; AStyle: Integer);
var i: Integer;
begin
  for i := 0 to 5 do
    if i = AStyle then ARec.StyleArray[i] := bf_Checked
    else ARec.StyleArray[i] := bf_Unchecked;
end;
```

Transferring the data

Once you've created a transfer buffer and filled in its values, getting that information into the dialog box is easy, because ObjectWindows handles it all for you. When you call *ExecDialog* to run the dialog box, it calls *TransferData* to copy the values from the transfer buffer to the individual control objects.

When you terminate the dialog box by clicking OK, *ExecDialog* transfers the values from the controls back into the transfer buffer before destroying the dialog box and its controls. Canceling the dialog box or closing it with the Control menu bypasses transferring the data back into the transfer buffer.

The transfer buffer, therefore, points to a persistent set of data, independent of the dialog box. Many times, a dialog box is created and destroyed many times while a program runs, but assigning its *TransferBuffer* field to the same data record each time allows the values of the controls to appear as they did when the previous dialog box closed.

Reading the return values

Reading the values back out of the transfer buffer is just the opposite of filling the buffer before executing the dialog box. Again, the *Pen* unit defines some functions to facilitate interpreting which of each group of radio buttons is checked:

```
function GetStyle(ARec: TPenData): Longint;
var i: Integer;
begin
  for i := 0 to 5 do
    if ARec.StyleArray[i] = bf_Checked then GetStyle := i;
end;
```

If the user cancels the dialog box, of course, you shouldn't bother reading the values: they are the same as those passed in. Normally, when you execute a dialog box with *ExecDialog*, you check the return value (*id_OK* if the user clicked OK, otherwise, *id_Cancel*) to see if the dialog box returned any useful data:

```
if Application^.ExecDialog(PenDlg) <> id_Cancel then
begin
  Val(PenData.XWidth, TempWidth, ErrorPos);
  SetAttributes(GetStyle(PenData), TempWidth, GetColorAttr(PenData));
end;
```

Calling the pen dialog box

To bring up a pen object's dialog box, call its *ChangePen* method. STEP06B.PAS does this in response to both the *cm_Pen* command generated by the Options | Pen menu choice and clicking the right mouse button:

These methods come from STEP06B.PAS.

```
procedure TStepWindow.CMPen(var Msg: TMessage);
begin
  CurrentPen^.ChangePen;     { CurrentPen is the window's pen object }
end;

procedure TStepWindow.WMRButtonDown(var Msg: TMessage);
begin
  if not ButtonDown then CurrentPen^.ChangePen;
end;
```

C H A P T E R

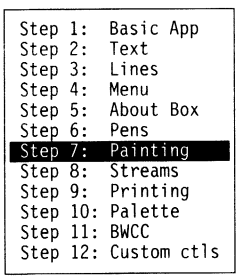
5

Repainting graphics

In the next three steps, you'll learn to

- Repaint the image on demand
- Save the image to a file and reload it
- Print the image

Step 7: Redisplaying graphics



You might be surprised to learn that the graphics and text you draw in a window using Windows functions like *TextOut* and *LineTo* disappear when you resize or uncover the window. Once the graphics data goes to Windows through calls to Windows functions, you can never get it back to redraw it.

To have a window redisplay its graphics, you've got to store the graphics (or the data to regenerate the graphics) in some type of structure, such as an object. With objects you can store simple or complex graphics polymorphically, and you can store the objects in fields of your window objects.

Painting vs. drawing

There are two ways to get an image into a window. The first, which you've been using, is *drawing*. When you draw, you create an image in real time, in response to user input. But the window

can't very well ask its user to recreate its graphics every time the screen gets updated.

Instead, you need to give your windows the ability to recreate their images on demand. Windows tells your window objects when they require updating or *painting*. The window must then somehow generate an image for display. ObjectWindows automatically calls the *Paint* method of your window type in response to the need for painting. The *Paint* inherited from *TWindow* does nothing. *Paint* is where you write the code to paint the contents of the window. In fact, *Paint* is called when the window first appears. *Paint* is responsible for updating the display with its current contents whenever needed.

There is one major difference between drawing graphics in the *Paint* method and at other times, such as in response to mouse actions. The display context to be used for painting is passed in the *PaintDC* parameter, so your program need not obtain or release it. You will, however, need to reselect your drawing tools into *PaintDC*.

To paint your window's contents, you are going to replay the actions that led to the original drawing on *DragDC*, but use *PaintDC* instead. The visual effect is the same as when they were drawn the first time by the user, much like replaying an audio recording of a concert. But first you need to store the graphics as objects, so you can paint them in a *Paint* method.

Storing graphics as objects

The drawings created with *Steps* are really just collections of varying numbers of lines. Every time you drag the mouse, you add another line. And each line is really just an arbitrary number of points, connected by drawn line segments. To store and reproduce such drawings, you need a flexible, expandable data type.

Collections are explained in detail in Chapter 19, "Collections."

The *TCollection* type, defined in the *Objects* unit, is perfectly suited to holding an unknown number of lines or points. Collection objects know how to grow dynamically as you add more elements. And collections don't know or care *what* they are collecting, so you can use the same mechanism for a drawing (a collection of lines) that you use for a line (a collection of points).

Conceptually, what you need to do is make the window aware of its contents so it can redraw them. The window contains a drawing, which is a collection of lines. You therefore need to:

- Give the window object a field to hold its collection of lines
- Define a line object that knows how to draw itself
- Update the mouse-message responses to add points to the stored lines

Adding an object field

To store the drawing as a collection of lines, add a field called *Drawing* to *TStepWindow*. At all times, *Drawing* contains the current drawing in the form of a collection of line objects. Whenever the window needs painting, it uses the data stored in *Drawing* to replay the drawing of the line.

Defining a line object

The next question to answer is, "What is a line?" You saw in Step 4 (page 32) that a drawn line is no more than a collection of points passed from Windows to the program through the *wm_MouseMove* message. You need object types to represent lines and points. Since a smart line-drawing object ought to be a reusable part, create a separate unit called *DrawLine* that defines line and point objects.

TLine contains all the information you need to draw a given line: a pen and a collection of points.

```
type
  PLine = ^TLine;
  TLine = object(TObject)
    Points: PCollection;
    LinePen: PPen;
    constructor Init(APen: PPen);
    constructor Load(var S: TStream);
    destructor Done; virtual;
    procedure AddPoint(AX, AY: Word);
    procedure Draw(ADC: HDC);
    procedure Store(var S: TStream);
  end;
```

LinePen simply points to a *TPen* object, and *Points* is a collection of point objects. *TLine* and its companion, *TLinePoint*, both have *Load* and *Store* methods that you'll take advantage of in Step 8 (page 59) to store your drawings on disk. Other than that, *TLine* is quite simple: the constructor and destructor create and dispose of

LinePen, *AddPoint* inserts a point object into *Points*, and *Draw* draws lines between each of the points in *Points*.

The *TLinePoint* object is even simpler:

```
type
  PLinePoint = ^TLinePoint;
  TLinePoint = object(TObject)
    X, Y: Integer;
    constructor Init(AX, AY: Integer);
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
  end;

constructor TLinePoint.Init(AX, AY: Integer);
begin
  X := AX;
  Y := AY;
end;
```

TLinePoint does not define any new behavior — it's just a data object to be used by *TLine* — but it needs to be an object in order to be stored on a stream later on (in Step 8, on page 59). Be sure to construct *Drawing* in *TStepWindow.Init* and dispose of it in *TStepWindow.Done*:

```
constructor TStepWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  ButtonDown := False;
  HasChanged := False;
  CommonPen := New(PPen, Init(ps_Solid, 1, 0));
  Drawing := New(PCollection, Init(50, 50));
end;

destructor TStepWindow.Done;
begin
  Dispose(CommonPen, Done);
  Dispose(Drawing, Done);
  inherited Done;
end;
```

To review, the main window of *Steps* holds a collection of lines in its *Drawing* field. As the user draws lines, you must convert them into objects and add them to *Drawing*. Then, when the window requires painting, you must iterate over *Drawing* and redraw each of its points.

Updating mouse methods

In order to store the lines as objects, you must alter *WMLButtonDown* and *WMMouseMove* to not just draw the lines but also store the point data in the line collection. Since more than one method will have to update the current line, add another field to *TStepWindow* called *CurrentLine*, of type *PLine*:

```
type
  TStepWindow = object(TWindow)
    CurrentLine: PLine;
      ⋮
  end;
```

In addition to drawing the line, *WMLButtonDown* creates a new line object each time it is called and adds it to the collection in *Drawing*. *WMMouseMove* just adds a new point to the end of the current line object as it draws the segments of the line in the window. By storing all the points of all the lines, your window retains the information needed to exactly reproduce your drawing.

```
procedure TStepWindow.WMLButtonDown(var Msg: TMessage);
begin
  if not ButtonDown then
  begin
    ButtonDown := True;
    SetCapture(HWindow);
    DragDC := GetDC(HWindow);
    CommonPen^.Select(DragDC);
    MoveTo(DragDC, Msg.lParamLo, Msg.lParamHi);
    CurrentLine := New(PLine, Init(CommonPen));
    Drawing^.Insert(CurrentLine);
  end;
end;

procedure TStepWindow.WMMouseMove(var Msg: TMessage);
begin
  if ButtonDown then
  begin
    LineTo(DragDC, Msg.lParamLo, Msg.lParamHi);
    CurrentLine^.AddPoint(Msg.LParamLo, Msg.LParamHi);
  end;
end;
```

You don't have to dispose of the old CurrentLine because it's stored in the collection Drawing. When you dispose of Drawing, it disposes of all the line objects.

WMLButtonUp requires no modification. You do need to dispose of all the line objects when clearing the drawing window, so add a *FreeAll* method call to *CMFileNew*:

```
procedure TStepWindow.CMFileNew(var Msg: TMessage);
begin
  Drawing^.FreeAll;
  InvalidateRect(HWindow, nil, True);
end;
```

Redrawing stored graphics

Iterator methods are explained in Chapter 19, "Collections."

This makes STEP07.PAS.

Now that *TStepWindow* is storing its current line, you must teach it to paint it on command, and this command is *Paint*. Let's write a *Paint* method for *TStepWindow* that repeats the actions of *WMLButtonDown*, *WMMouseMove*, and *WMLButtonUp*. By iterating over the collection of lines, telling each one to draw itself, *Paint* recreates the drawing just as you drew it. Here is the *Paint* method:

```
procedure TStepWindow.Paint(PaintDC: HDC; var PaintInfo:
  TPaintStruct);

  procedure DrawLine(P: PLine); far;
  begin
    P^.Draw(PaintDC);
  end;

begin
  Drawing^.ForEach(@DrawLine);
end;
```

The line object's *Draw* method also uses the *ForEach* iterator to draw lines between each of its points:

```
procedure TLine.Draw(ADC: HDC);
var First: Boolean;

  procedure DrawLine(P: PLinePoint); far;
  begin
    if First then MoveTo(ADC, P^.X, P^.Y)
    else LineTo(ADC, P^.X, P^.Y);
    First := False;
  end;

begin
  First := True;
  LinePen^.Select(ADC);
  Points^.ForEach(@DrawLine);
  LinePen^.Delete;
end;
```

Step 8: Storing the drawing in a file

```
Step 1:  Basic App
Step 2:  Text
Step 3:  Lines
Step 4:  Menu
Step 5:  About Box
Step 6:  Pens
Step 7:  Painting
Step 8:  Streams
Step 9:  Printing
Step 10: Palette
Step 11: BWCC
Step 12: Custom ctls
```

Now that you've stored a data representation of the drawing as part of the window object, you can easily write that data to a file (actually, a buffered DOS stream) and read it back.

In this step, you'll add object fields to store the saving status and modify the file saving and opening methods. Using the stream objects supplied with ObjectWindows, you'll find that it's convenient to store objects and their data to a file.

Monitoring file status

There are two characteristics of the drawing you need to monitor. You already track whether the file has changed (the *HasChanged* field you added in Step 1, on page 8), but now you need to know if there is a file currently loaded. Like *HasChanged*, *IsNewFile* is a Boolean attribute of *TStepWindow*, so make it a field, too:

```
TStepWindow = object(TWindow)
  ButtonDown, HasChanged, IsNewFile: Boolean;
    :
end;
```

HasChanged is *True* if the current drawing is "dirty." Dirty means the drawing has changed since it was last saved, or it has never been saved. You already set *HasChanged* to *True* when the user starts drawing, and set it to *False* when the window gets cleared. When the user opens a new file or saves the existing one, you should set *HasChanged* to *False*.

IsNewFile indicates that the drawing did not come from a file, so saving the drawing will require the user to name a file. *IsNewFile* is *True* only when the application first starts and after the user selects the File | New menu. It is set to *False* whenever a file is opened or saved. In fact, *FileSave* uses *IsNewFile* to see if the file can be saved immediately or if the user needs to select a file from a file dialog.

Listed here are the file-saving and file-loading methods. At this point, they do everything but save and load files. The file saving behavior has been concentrated into one new method, called *WriteFile*, and file opening is delegated to *ReadFile*.

This makes STEP08A.PAS.

```
procedure TStepWindow.CMFileNew(var Msg: TMessage);
begin
  if CanClose then
  begin
    Drawing^.FreeAll;
    InvalidateRect(HWindow, nil, True);
    HasChanged := False;
    IsNewFile := True;
  end;
end;

procedure TStepWindow.CMFileOpen(var Msg: TMessage);
begin
  if CanClose then
    if Application^.ExecDialog(New(PFileDialog, Init(@Self,
      PChar(sd_FileOpen), StrCopy(FileName, '*.PTS')))) = id_Ok then
      ReadFile;
end;

procedure TStepWindow.CMFileSave(var Msg: TMessage);
begin
  if IsNewFile then CMFileSaveAs(Msg) else WriteFile;
end;

procedure TStepWindow.CMFileSaveAs(var Msg: TMessage);
begin
  if IsNewFile then StrCopy(FileName, '');
  if Application^.ExecDialog(New(PFileDialog,
      Init(@Self, PChar(sd_FileSave), FileName))) = id_Ok then
    WriteFile;
end;

procedure TStepWindow.ReadFile;
begin
  MessageBox(HWindow, @FileName, 'Load the file:', mb_OK);
  HasChanged := False;
  IsNewFile := False;
end;

procedure TStepWindow.WriteFile;
begin
  MessageBox(HWindow, @FileName, 'Save the file:', mb_OK);
  HasChanged := False;
  IsNewFile := False;
end;
```

Saving and loading Files

For more information on using streams with your objects, see Chapter 20, "Streams."

Now that you've built the framework for saving and loading files, all that is left is to actually save and load the collection of points into a file. For this, you use the automatic object-storing mechanism of streams. First, you have to teach the line and point objects to store and load themselves (since collections already know how). Then you modify the *WriteFile* and *FileOpen* methods to make use of streams.

Here is the code necessary for teaching *TLine* and *TLinePoint* objects to store and load themselves:

```
const
  RLinePoint: TStreamRec = (
    ObjType: 200;
    VmtLink: Ofs(TypeOf(TLinePoint)^);
    Load: @TLinePoint.Load;
    Store: @TLinePoint.Store);

  RLine: TStreamRec = (
    ObjType: 201;
    VmtLink: Ofs(TypeOf(TLine)^);
    Load: @TLine.Load;
    Store: @TLine.Store);

constructor TLinePoint.Load(var S: TStream);
begin
  S.Read(X, SizeOf(X));
  S.Read(Y, SizeOf(Y));
end;

procedure TLinePoint.Store(var S: TStream);
begin
  S.Write(X, SizeOf(X));
  S.Write(Y, SizeOf(Y));
end;

constructor TLine.Load(var S: TStream);
begin
  Points := PCollection(S.Get);
  LinePen := PPen(S.Get);
end;

procedure TLine.Store(var S: TStream);
begin
  S.Put(Points);
  S.Put(LinePen);
end;
```

```
procedure StreamRegistration;
begin
  RegisterType(RCollection);
end;
```

You must call *StreamRegistration* (which is in *Steps*) to register *TCollection* when the application starts up. You can put this call in the *TStepWindow.Init* method. The *DrawLine* unit registers *TLinePoint* and *TLine* in its initialization code, so just by adding *DrawLine* to your **uses** clause, lines and points are registered.

The final step is to rewrite the *WriteFile* and *ReadFile* methods to actually write to and read from the stream:

This brings you up to STEP08B.PAS.

```
procedure TStepWindow.ReadFile;
var
  TempColl: PCollection;
  TheFile: TDosStream;
begin
  TheFile.Init(FileName, stOpen);
  TempColl := PCollection(TheFile.Get);
  TheFile.Done;
  if TempColl <> nil then
  begin
   Dispose(Drawing, Done);
   Drawing := TempColl;
   InvalidateRect(HWindow, nil, True);
  end;
  HasChanged := False;
  IsNewFile := False;
end;

procedure TStepWindow.WriteFile;
var
  TheFile: TDosStream;
begin
  TheFile.Init(FileName, stCreate);
  TheFile.Put(Drawing);
  TheFile.Done;
  IsNewFile := False;
  HasChanged := False;
end;
```

Step 9: Printing the image

```
Step 1:  Basic App
Step 2:  Text
Step 3:  Lines
Step 4:  Menu
Step 5:  About Box
Step 6:  Pens
Step 7:  Painting
Step 8:  Streams
Step 9:  Printing
Step 10: Palette
Step 11: BWCC
Step 12: Custom ctls
```

Printing from Windows can be a daunting task, but ObjectWindows provides a simple mechanism to add printing to your Windows applications.

Adding printing takes only three steps:

- Constructing a printer object
- Creating a printout object
- Printing the printout object

Constructing a printer object

TPrinter is defined in the OPrinter unit, so be sure to add OPrinter to your ***uses*** *clause.*

Any ObjectWindows program can access a printer by using an object of type *TPrinter*. In this case, your application's main window should construct a printer object and store it in an object field called *Printer*:

```
constructor TStepWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
   :
  Printer := New(PPrinter, Init);
end;
```

This is all you normally have to do to initialize a printer object. By default, *TPrinter* uses the default printer specified in your WIN.INI file. *TPrinter* also provides a mechanism for selecting alternate printers.

Creating a printout object

ObjectWindows handles printed output in much the same way as it handles screen output. That is, instead of writing directly to the output device (or even directly to Windows), you direct your output to an object that knows how to present its information to the output device. With window objects, this means using the *Paint* method. For printed output, you use objects derived from the abstract object type *TPrintout* and a method called *PrintPage*.

Printing documents is explained fully in Chapter 15, "Printing objects."

There are two basic cases you'll need to deal with in generating printed output from Windows applications: printing documents and printing window contents. Printing the contents of a window is easiest, since windows already know how to represent

themselves. As it turns out, however, this is just a special case of printing a document.

Writing to a device context

Up to this point, you have always written text and graphics to display contexts, which is the way Windows represents the client area of its windows. A display context is just a specialized version of a *device context*, which is the mechanism through which Windows applications deal with various devices, such as screens, printers, or communications devices.

Luckily, writing to one device context is much like writing to another, so it is quite simple to tell a window object, for example, to use its *Paint* mechanism to write to a printer.

Creating printed output from a window

TWindowPrint is a special descendant of *TPrintout*, designed to print the contents of a window. Printing window contents is easy for two reasons: you're only dealing with a single page, and the window object already knows how to paint its image.

Normally, when printing a document, your application needs to iterate through the printing process for each page in the document. This is not necessary for printing a window, since the window has only a single image, as it appears on the screen.

Printing the contents is therefore as simple as directing the *Paint* method of the window object to paint onto a device context that suits your printer, rather than one in a window. For example, *TWindowPrint.PrintPage* just calls the window object's *Paint* method:

```
procedure TWindowPrint.PrintPage(DC: HDC; Page: Word; Size: TPoint;
  var Rect: TRect; Flags: Word);
var PS: TPaintStruct;
begin
  Window^.Paint(DC, PS);
end;
```

Because the *DC* parameter passed to *PrintPage* already points to a device context suitable for the printer, all *PrintPage* has to do for this simple case is tell the window object to paint its contents onto that device context. Now your printout object also knows how to represent itself.

Printing the printout

Once you have a printout object that knows how to represent itself, all that's left is to actually send the printout to the printer. *Steps* does this in response to the command *cm_FilePrint*, generated by the Print command on the File menu:

```
procedure TStepWindow.CMFilePrint(var Msg: TMessage);
var P: PPrintout;
begin
  if IsNewFile then StrCopy(FileName, 'Untitled');
  P := New(PWindowPrint, Init(FileName, @Self));
  Printer^.Print(@Self, P);
  Dispose(P, Done);
end;
```

Very simply, *CMFilePrint* constructs a printout object titled with a given name (the name of the points file or 'Untitled') and filled with the contents of itself (since this is the only window in the application).

Once the printout object exists, *CMFilePrint* tells the printer object to print it, attaching any error messages or dialog boxes to that window (hence the *@Self* parameter). When printing finishes, *CMFilePrint* disposes of the printout object.

Picking a different printer

One of the features of *TPrinter* is the ability to change the printer setup. *TPrinter* defines a *Setup* method that displays a printer setup dialog box that enables the user to pick a printer from those installed in Windows and also provides access to the printer device's configuration dialog box.

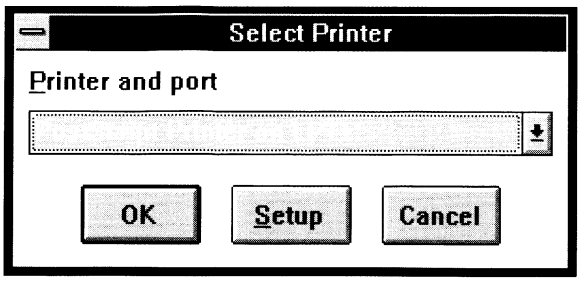
To display the printer setup dialog box, your application calls the printer object's *Setup* method. *Steps* does this in response to the command *cm_FileSetup*:

This makes STEP09.PAS.

```
procedure TStepWindow.CMFileSetup(var Msg: TMessage);
begin
  Printer^.Setup(@Self);
end;
```

The printer setup dialog box is an instance of type *TPrinterSetupDlg*, as shown in Figure 5.1.

Figure 5.1
The printer setup dialog box



All printers installed in WIN.INI appear in the printer setup dialog box's combo box. This gives users access to all installed printers.

C H A P T E R

6

Popping up a window

You have now created two types of windows, main windows (the *TStepWindow* object) and modal child windows that get created and destroyed each time you need them (the About box and various message boxes). But full-featured Windows programs often need to keep child windows active for indefinite periods of time. One example is the SpeedBar in the Windows-hosted IDE.

The child windows in *Steps* up to now are all fixed in size and created from resource templates. In Steps 10 through 12 you'll

- Create a dynamically-sized window that stays around for the duration of the program
- Add custom controls to the window
- Create an interactive window of your own

Finally, after the last step (starting on page 85) you'll find suggestions of some ways to extend *Steps* further.

Step 10: Adding a pop-up window

```
Step 1:  Basic App
Step 2:  Text
Step 3:  Lines
Step 4:  Menu
Step 5:  About Box
Step 6:  Pens
Step 7:  Painting
Step 8:  Streams
Step 9:  Printing
Step 10: Palette
Step 11: BWCC
Step 12: Custom ctls
```

Creating and destroying windows and dialog boxes works fine for windows and dialog boxes that aren't used often. But in some cases, you want a child window available most of the time, such as with a tool palette.

In this step, you'll do the following:

- Add a field to the main window
- Construct a floating pen palette
- Show and hide the pen palette

Figure 6.1 shows the pen palette window that appears when the user chooses Palette | Show.

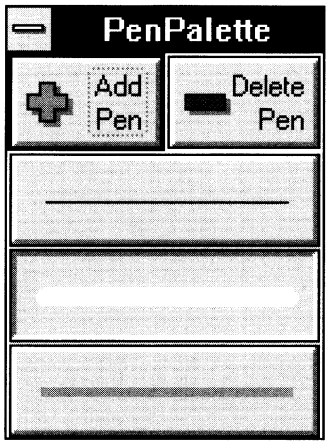


Figure 6.1
Steps' pen palette with three pens

Since this is the first "new" window you've added, other than the main window, which is automatically displayed, this is a good time to look at how window objects and window elements are created and displayed.

Adding a child to a window

The modal child windows you've used up to now have been easy to manage. You create them, use them, and dispose of them. But a child window that isn't modal needs to be managed. For example, it must close if the application closes and hide itself if the main window gets minimized.

Most child window behavior, such as closing and hiding, is automatic. The only time you have to do anything special is if you

want the child window to do something independent of its parent. For example, the palette window you're about to add will be able to hide or show itself while the main window stays visible. Windows that can move or display themselves separately from their parent windows are called *independent* child windows. In the next steps you'll create dependent windows attached to the independent palette window.

Because the main window has to send commands to the palette window, it needs a pointer to that window, so add a field to *TStepWindow* for its pen palette. *TStepWindow* now has these fields:

```
TStepWindow = object(TWindow)
  DragDC: HDC;
  ButtonDown: Boolean;
  FileName: array[0..fsPathName] of Char;
  HasChanged, IsNewFile: Boolean;
  Drawing: PCollection;
  CurrentLine: PDLine;
  Printer: PPrinter;
  PenPalette: PPenPalette;                    { the palette window }
    ⋮
end;
```

All that remains is to construct a child window object and assign it to *PenPalette*, and that is the subject of the next section.

Constructing the palette window

Child-window objects are normally constructed in their parent windows' constructors. Just as you would initialize any other field, you assign values to any pointers to child windows. In this case,

```
constructor TStepWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
    ⋮
  PenPalette := New(PPenPalette, Init(@Self, 'Pen Palette'));
end;
```

Assigning the parent window

Parent windows automatically manage their children, so when you create a child window, you pass it a pointer to its parent-window object. Since the parent window usually constructs its child windows, the parent-window pointer is normally *@Self*.

The chief exception is the application's main window. Since it has no parent, you pass **nil** to the main window's constructor.

Every window object has a list of its child windows, which ensures that each child window gets created, shown, hidden, and closed at the appropriate times. Constructing and destructing child windows updates the child window list automatically: constructing a child-window adds it to its parent's list; destructing it removes it from the list.

Creating the screen element

One of the things ObjectWindows handles for you when you construct a child-window object is creating the screen element that corresponds to your object. This is the converse of what you did with *InitResource* in Step 6 (page 43). Then, you had a screen element created from a resource, and you associated an object with it so you could manipulate the screen element. Now you have created your own object, and you need to tell Windows to create a corresponding screen element.

MakeWindow and creating screen elements are described in detail in Chapter 9, "Interface objects."

When you did this for a dialog box in Step 3 (page 23), you called *ExecDialog*, a *TApplication* method that created a screen element and executed a modal dialog box. The corresponding method for non-modal (or modeless) dialog boxes and windows is *TApplication.MakeWindow*. The biggest differences are that *MakeWindow* doesn't automatically display the screen element it creates, and it doesn't enter a modal state.

The process of constructing and displaying a window, then, is usually a three-part process:

- Construct the window object with *Init*
- Create a screen element with *MakeWindow*
- Show the window with *Show*

Luckily, the second and third steps are done automatically for the application's main window. In addition, calling *MakeWindow* for a parent window automatically calls *MakeWindow* for any windows in its child-window list, so child windows of the main window (such as the pen palette) also get screen elements automatically.

You'll show the child window in the next section.

Showing and hiding the palette

Child windows other than dialog boxes are created and shown by default by their parents. (This process is controlled by the

EnableAutoCreate and *DisableAutoCreate* methods in every interface object.) But you might want to show or hide a child window on command. Both are functions of the *Show* method inherited from *TWindowsObject*.

Show either shows or hides the window, depending on the parameter passed to it. The parameter is one of the *sw_* constants defined by Windows. *TStepWindow* calls the pen palette's *Show* method with different parameters in response to the menu commands Palette | Show and Palette | Hide, which generate the commands *cm_PalShow* and *cm_PalHide*, respectively:

This completes STEP10.PAS.

```
procedure TStepWindow.CMPalShow(var Msg: TMessage);
begin
  PenPalette^.Show(sw_ShowNA);
end;

procedure TStepWindow.CMPalHide(var Msg: TMessage);
begin
  PenPalette^.Show(sw_Hide);
end;
```

You can easily specify the child window you want to act on if you have a field in the parent window pointing to the child window you want to manipulate.

Step 11: Adding custom controls

```
Step 1:  Basic App
Step 2:  Text
Step 3:  Lines
Step 4:  Menu
Step 5:  About Box
Step 6:  Pens
Step 7:  Painting
Step 8:  Streams
Step 9:  Printing
Step 10: Palette
Step 11: BWCC
Step 12: Custom ctls
```

In Step 10 (page 68) you added an independent child window to the main window. Now you'll add dependent child windows called *controls*. These controls have the pen palette window as their parent. Remember that the pen palette window is an independent child window whose parent is the application's main window. Therefore, the pen palette is both a child window of the main window and a parent window of the controls.

You have already dealt with controls and control objects in Step 6 (page 43), but you just associated objects with controls defined in a resource. Constructing a control at run time requires a bit more work, since you have to specify the position and size of the controls as well as the type.

The pen palette shown in Figure 6.1 uses two custom push buttons and a series of bitmapped pictures that look and act much like buttons, each representing a pen you can use in your drawing. These pen "buttons" are not actually controls, but rather

images on a single child window that knows how to identify where you click the mouse.

In this step, you will add the fancy bitmapped buttons by

- Adding simple button controls
- Enabling custom controls in the program
- Defining bitmaps for the buttons

The ObjectWindows type *TControl* supplies the behaviors of all controls in general, and its descending types handle each type of control. For example, *TListBox* defines list box objects and *TEdit* defines edit control objects. You should also realize that *TControl* descends from *TWindow*.

Adding buttons to the palette

While they behave identically, there's an important programming difference between the controls in dialog boxes, such as file dialog boxes, and the controls in windows, such as your palette window. You specify the controls of a dialog box in the dialog box's resource. They are not objects and the dialog boxes that own them are completely responsible for managing them. Chapter 11 shows how to create your own dialog boxes from dialog resources and to manage their controls.

A window's controls, however, are specified by object definitions. A parent window manages its controls through the methods defined by the ObjectWindows control objects. For example, to get the text item the user has selected from a list box, call the list box object's *GetSelString* method. Like a window or dialog box object, a control object has a corresponding visual element.

A control object and its control element are linked through the control object's *ID* field. Each control has a unique ID that is used by its parent window to identify the control for routing of control events, such as when the user clicks on a button. For clarity, you should define constants for each control ID:

```
const
  id_Add = 101;
  id_Del = 102;
  MaxPens = 9;
```

MaxPens sets the maximum number of pens the palette will hold. Nine fit nicely on a standard VGA screen.

Control objects as fields

As with other child windows, it is often convenient to store a pointer to a control object as a field in a window object. This is only necessary for child windows that will later be manipulated directly by calling their object methods. In this case, both of the buttons qualify. *TPenPalette* stores each of these control objects in a separate field. Here is part of *TPenPalette*'s object declaration:

```
TPenPalette = object(TWindow)
  AddBtn, DelBtn: PButton;
   ⋮
end;
```

Once these child control objects have been instantiated, you can manipulate them with method calls. For example, at appropriate times you want to enable or disable the buttons by calling their *Enable* and *Disable* methods. It *is* possible to access the control objects for child windows not stored as fields by using the *ChildList* of the parent window, but it is far more convenient to do so with fields.

Managing controls

Any window type that has control objects (or any other child windows) must define a constructor, *Init*, to construct its control objects. In addition, it can override *SetupWindow*, to set up the controls prior to display. The parent window (*TPenPalette*) automatically creates and displays all of its child windows.

Listed here is the pen palette's *Init* method. The first thing it does is set its own location and size attributes. Since a window's *Init* method is responsible for setting its own creation attributes, and because its controls are created with it, you must also construct its controls in its *Init*. A parent window (*@Self*) is the first parameter in every control's constructor call, followed by a control ID.

```
constructor TPenPalette.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  with Attr do
  begin
    Style := Style or ws_Tiled or ws_SysMenu or ws_Visible;
    W := 133;
    H := GetSystemMetrics(sm_CYCaption) + 42;
  end;
```

```
    AddBtn := New(PButton, Init(@Self, id_Add, 'Add Pen', 0, 0,
      65, 40, True));
    DelBtn := New(PButton, Init(@Self, id_Del, 'Del Pen', 65, 0,
      65, 40, False));
  end;
```

Whenever you override a window's SetupWindow method, be sure to call the inherited SetupWindow first, because it creates all of the child controls.

The virtual method *TPenPalette.SetupWindow* is called following the window's creation in order to set up the window's controls. Since you're only dealing with buttons here, there's no initialization necessary, but *TPenPalette.SetupWindow* initially disables one of the buttons. If you were using another control, such as a list box, you might need to use *SetupWindow* to initialize the control object.

Init and *SetupWindow* methods are enough to get all the controls displayed properly in the palette window. The buttons will press, but with no resulting action. In Step 12 (page 76), you'll define a response to control events.

Hiding instead of closing

If you double-click the system-menu box of the pen palette, it goes away permanently. Choosing Palette | Show can no longer show the palette because the object and its screen element have been destroyed. There's nothing to show. You can override this by adding a *CanClose* method that hides the window and then refuses to close:

This makes STEP11A.PAS.

```
function TPenPalette.CanClose: Boolean;
begin
  Show(sw_Hide);
  CanClose := False;
end;
```

Now double-clicking on the system-menu box hides the window but doesn't close it, so you can show it again later.

Normally, having a child window that always returns *False* from *CanClose* could keep your whole application from ever closing. But *TStepWindow* doesn't check its child windows before closing, because you overrode its *CanClose* method in Step 1 (page 8).

Enabling custom controls

By now you have observed that control objects create standard Windows controls. The *TButton* objects you just created, for instance, resulted in standard gray buttons in the palette window. The IDE and the dialog boxes you've created from resources, however, use fancier buttons with bitmapped images on them.

ObjectWindows gives you an easy way to use these same custom controls in your programs.

Enabling the use of Borland Windows Custom Controls (BWCC) is as easy as using one unit. Simply add *BWCC* to the **uses** clause of your main program. This has two immediate effects. The first is that all standard dialog boxes (such as the file dialog boxes you've already added to *Steps*) use custom controls instead of standard controls for common items such as OK and Cancel buttons, check boxes, and radio buttons. The second is that all controls created from objects are custom controls rather than standard controls.

Use and design of Borland Windows Custom Controls is explained in Chapter 12, "Control objects."

In fact, once you have added *BWCC* to the **uses** clause in *Steps*, go ahead and recompile the program and access the file dialog boxes. With no other effort, you have significantly upgraded the look of your user interface.

But what about the buttons in the pen palette? They were created from control objects with *BWCC* used in the program, but they look just like normal buttons. The answer, of course, is that you haven't yet defined bitmaps for the buttons, so by default, they simply use the labels passed in the *Init* constructor. In the next section, you'll see how to add bitmaps to your custom controls.

Creating bitmaps for buttons

Although you can create bitmaps for custom controls at run time, this is the sort of task that should be done with resources. Using Resource Workshop, you need to create three different bitmaps for each button: one for the up position, one for the down position, and one for the up position with the input focus.

There are no restrictions on what you can do with these bitmaps. You could change the colors of images depending on the button state, or you could move the images (the usual button behavior) when pressed, and add a dotted line around the button text when it has the focus. Figure 6.2 shows the three bitmaps for the Add Pen button in the pen palette.

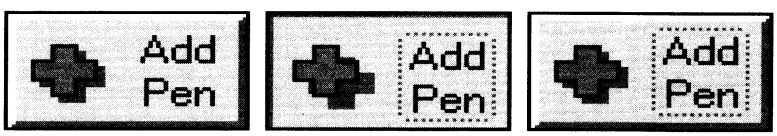


Figure 6.2
Bitmapped images for a custom button

Numbering the bitmap resources

For EGA systems, the resources used are 2000 + the ID, 4000 + the ID, and 6000 + the ID, respectively.

The only tricky part of defining bitmaps for your buttons is assigning the resource IDs. BWCC controls know which bitmaps to use, based on the control ID of the particular control. For buttons in VGA systems, the resources used are 1000 + the ID for the "up" image, 3000 + the ID for the "down" image, and 5000 + the ID for the focused image.

Since the Add Pen button has an ID of 101 (*id_Add*), enabling BWCC causes the button to look for resources 1101, 3101, and 5101 for its three bitmaps. STEP11B.PAS adds the compiler directive

This completes STEP11B.PAS.

```
{$R PENPAL.RES}
```

to access customized bitmaps for the Add Pen and Del Pen buttons.

Step 12: Creating a custom window control

```
Step 1:  Basic App
Step 2:  Text
Step 3:  Lines
Step 4:  Menu
Step 5:  About Box
Step 6:  Pens
Step 7:  Painting
Step 8:  Streams
Step 9:  Printing
Step 10: Palette
Step 11: BWCC
Step 12: Custom ctls
```

The really fun part of creating this pen palette comes in creating your own custom palette window. Here you finally go beyond those abilities provided by the windows standard tools and create something.

In this step, you'll

- Make the palette window dynamically resize itself
- Respond to notification messages from controls
- Create a multipane palette object

Dynamically resizing the palette

Since each pen you store on the palette is the same size (40 pixels high and 128 pixels wide), you need to make sure the palette window can grow and shrink by that amount each time you add or remove a pen. The *TPenPalette* object defines two methods to deal with this: *Grow* and *Shrink*.

```
procedure TPenPalette.Grow;
var WindowRect: TRect;
begin
  GetWindowRect(HWindow, WindowRect);
```

```
  with WindowRect do
    MoveWindow(HWindow, left, top, right - left,
      bottom - top + 40, True);
end;

procedure TPenPalette.Shrink;
var WindowRect: TRect;
begin
  GetWindowRect(HWindow, WindowRect);
  with WindowRect do
    MoveWindow(HWindow, left, top, right - left,
      bottom - top - 40, True);
end;
```

Both methods find out the coordinates of the boundaries of the window, modify them, and tell the window to use the new coordinates for its boundaries. The *GetWindowRect* API function returns a *TRect* structure containing the top, bottom, left, and right coordinates of the window. *Grow* adds 40 pixels to the bottom of the window, and *Shrink* subtracts the same amount.

In the next section, you'll call the *Grow* and *Shrink* methods in response to pressing the Add Pen and Del Pen buttons.

Responding to control events

The main difference between the palette window and the modal dialog boxes you've used before this is that in a modal dialog box, you manipulate the controls and then read the results at the end *if* the user clicks OK. In this modeless palette window, you are dealing with an active, dynamic part of the program, and you can actively respond to every control when it gets used.

At this point, the buttons appear in the palette window, but clicking the buttons has no effect. That's because clicking and selecting are control events. They are very similar to the menu events you responded to in Step 4 (page 32).

Command messages and control notifications are explained in detail in Chapter 16, "Windows messages."

You responded to menu events by defining command-response methods, and you do something similar with control messages. Control events produce child-ID-based messages, which are like command messages but carry the control ID instead of the menu ID. Use the sum of a control's ID and the constant *id_First* to identify a child-ID-based method header.

Naming control-response methods

As with message-response methods, which you named after the messages, child-ID-based methods should also be named after the ID messages. Since the two buttons you want to respond to have IDs of *id_Add* and *id_Del*, *TPenPalette* needs methods called *IDAdd* and *IDDel*:

```
TPenPalette = object(TWindow)
  AddBtn, DelBtn: PBitButton;
  constructor Init(AParent: PWindowsObject; ATitle: PChar);
  procedure Grow;
  procedure SetupWindow; virtual;
  procedure Shrink;
  procedure IDAdd(var Msg: TMessage); virtual id_First + id_Add;
  procedure IDDel(var Msg: TMessage); virtual id_First + id_Del;
end;
```

Now all you have to do is define the *IDAdd* and *IDDel* methods to perform the appropriate actions to respond to their respective buttons. For now, *IDAdd* should just make the window grow, and *IDDel* should make the window shrink:

This completes STEP12A.PAS.

```
procedure TPenPalette.IDAdd(var Msg: TMessage);
begin
  Grow;
end;

procedure TPenPalette.IDDel(var Msg: TMessage);
begin
  Shrink;
end;
```

Adding the palette "buttons"

Now that you have a palette window, all you need is a way to display and choose the pens in the palette. For this, you can use a relatively simple descendant of *TWindow* and a collection of pen objects.

In this section, you'll

- Define the pen palette object
- Display bitmaps on the palette
- Select pens based on mouse clicks

Defining the palette object

Because the pen palette window can change its size, the palette within the window can actually stay fixed. You can take advantage of the clipping capabilities of Windows to show only the part of the palette that has pens drawn on it.

The palette itself only needs a few data fields: a collection of pens, an indication of which pen is currently selected, and handles for bitmap images to represent the pens. The bitmap handles are private fields, not because they need to be secret, but rather to keep other code from inadvertently altering them.

Here is the declaration of the palette object:

```
TPenPic = object(TWindow)
  PenSet: PCollection;
  CurrentPen: Integer;
  constructor Init(AParent: PWindowsObject);
  destructor Done; virtual;
  procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
    virtual;
  procedure AddPen(APen: PPen);
  procedure DeletePen;
  procedure SetupWindow; virtual;
  procedure WMLButtonDown(var Msg: TMessage);
    virtual wm_First + wm_LButtonDown;
private
  UpPic, DownPic: HBitmap;
end;
```

TPenPic doesn't need very many methods. It has a simple constructor to create the collection of pens and a destructor to get rid of them. The *SetupWindow* method just moves the location of the palette within its parent window. *AddPen* and *DeletePen* insert and remove pens from the collection, and *WMLButtonDown* interprets mouse clicks to select pens from the palette. Finally, *Paint* draws the "buttons" that represent the pens in the collection.

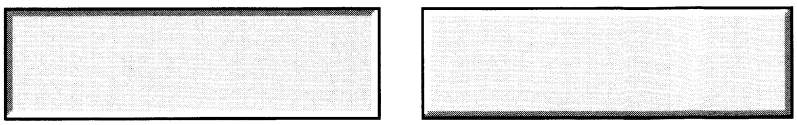
Notice also that *TPenPic* is a descendant of *TWindow*, not *TControl*. Although your new object behaves much like a Windows control, you need to derive it from *TWindow* because *TControl* only works with Windows' standard controls, such as push buttons and scroll bars. When you create your own controls, you start with *TWindow*.

Creating and destroying the palette

Constructing and destructing the palette object is rather simple. *Init* calls *TWindow.Init*, then changes the window's style to be a visible child window. *PenSet* is initialized as a collection of fixed size, large enough to hold the maximum number of pens specified by the *MaxPens* constant, and not increasing. The currently selected pen, *CurrentPen*, is set to –1, meaning that no pen is selected.

Finally, *Init* loads two bitmap images into *UpPic* and *DownPic*. These serve as background bitmaps for each pen in the palette. *DownPic* is painted behind the selected pen, *UpPic* behind the others. The two bitmaps are shown in Figure 6.3.

Figure 6.3
The pen palette background bitmaps



The *Done* destructor disposes of the bitmaps and the pen collection before calling its inherited *Done*. Calling *DeleteObject* to get rid of the bitmaps is very important. Like display contexts, bitmaps are Windows resources, managed in Windows limited memory. If you bring them into Windows memory and fail to delete them, your program (and other programs running concurrently) will lose access to that memory.

Here are the listings of the palette object's *Init* and *Done*:

```
constructor TPenPic.Init(AParent: PWindowsObject);
begin
  inherited Init(AParent, nil);
  Attr.Style := ws_Child or ws_Visible;
  PenSet := New(PCollection, Init(MaxPens, 0));
  CurrentPen := -1;
  UpPic := LoadBitmap(HInstance, 'PAL_UP');
  DownPic := LoadBitmap(HInstance, 'PAL_DOWN');
end;

destructor TPenPic.Done;
begin
  DeleteObject(UpPic);
  DeleteObject(DownPic);
  Dispose(PenSet, Done);
  inherited Done;
end;
```

Fitting into a parent window

You might have noticed that *TPenPic* didn't specify its position in its constructor as most controls would. The reason is that you can't be sure of the coordinates of your window until that window actually exists. *TPenPalette* gets around this when creating its button objects by calling the API function *GetSystemMetrics* to find the height of the window caption bar. Although this might work, there is also another approach.

Rather than positioning the palette to a particular place within the parent window, you really want to place it at a certain position within the parent's client area. That way, if you add a menu or change the frame or otherwise alter the outside of the window, your palette object will still be properly positioned.

The repositioning of the window takes place in *SetupWindow*, because the palette window needs to use the parent window's window handle, which is not available until after its own *SetupWindow* has been called. A child window can safely assume that its parent has a valid handle, however, once it reaches *SetupWindow*.

```
procedure TPenPic.SetupWindow;
var ClientRect: TRect;
begin
  inherited SetupWindow;
  GetClientRect(Parent^.HWindow, ClientRect);
  with ClientRect do
    MoveWindow(HWindow, 1, bottom - top + 1, 128,
      40 * MaxPens, False);
end;
```

TPenPic uses the Windows API function *GetClientRect* to return the coordinates of the palette window's client area. It then repositions itself with *MoveWindow* to a position just below the button objects, setting its height long enough to accommodate all the pens the pen collection could hold. Note that the last parameter to *MoveWindow* is a Boolean value indicating whether the window should be repainted after moving. Since the palette hasn't even been displayed yet, there is no reason to repaint it, so *TPenPic.SetupWindow* passes *False*.

Adding and removing pens

In the last section, you responded to messages from the Add Pen and Del Pen buttons by changing the size of the palette window. Now it's time to change that response to actually add and delete pens from the palette, which in turn tells the palette window to change size. Instead of calling its own *Grow* and *Shrink* methods, then, *TPenPalette*'s *IDAdd* and *IDDel* should call the palette object's *AddPen* and *DeletePen*, respectively:

```
procedure TPenPalette.IDAdd(var Msg: TMessage);
begin
  Pens^.AddPen(CommonPen);
end;

procedure TPenPalette.IDDel(var Msg: TMessage);
begin
  Pens^.DeletePen;
end;
```

AddPen takes the pen it's passed, copies it into the collection, and marks that pen as the currently selected pen. It then enables the Del Pen button in the pen palette window, disables the Add Pen button if the collection is full, and tells the parent window to grow enough to show the new pen.

```
procedure TPenPic.AddPen(APen: PPen);
begin
  CurrentPen := PenSet^.Count;
  with APen^ do PenSet^.Insert(New(PPen, Init(Style, Width, Color)));
  with PPenPalette(Parent)^ do
  begin
    DelBtn^.Enable;
    if PenSet^.Count >= MaxPens then
      AddBtn^.Disable;
    Grow;
  end;
end;
```

TPenPic can typecast its Parent field to take advantage of features specific to TPenPalette. Most window objects aren't this tightly bound to a particular parent type, and so shouldn't assume anything about the type of their parent windows.

DeletePen essentially reverses all the actions of *AddPen*. If there is a pen selected in the palette, it removes it from the collection and packs the collection so all active pens are contiguous. It then indicates that no pen is currently selected (because the selected pen has just been removed) and disables the Del Pen button, because Del Pen acts on the selected pen. Next, it enables the Add Pen button, because deleting a pen automatically makes room to add at least one more pen. Finally, it tells the parent window to reduce its size, because there are fewer pens to show.

```
procedure TPenPic.DeletePen;
begin
  if CurrentPen > -1 then
  begin
    PenSet^.AtFree(CurrentPen);
    PenSet^.Pack;
    CurrentPen := -1;
    with PPenPalette(Parent)^ do
    begin
      AddBtn^.Enable;
      DelBtn^.Disable;
      Shrink;
    end;
  end;
end;
```

Notice that both *AddPen* and *DeletePen* take advantage of the fact that the pen palette window has pointers to its buttons for easy communication with their methods. If *TPenPalette* didn't have *AddBtn* and *DelBtn* as fields, the palette object would have to either look for them by their IDs and then send them messages, or would have to send a message to the parent window, which would in turn communicate with the buttons somehow.

Drawing the palette entries

All the design that has gone into the palette object really pays off when it comes time to display the palette entries. Because the pens are stored in a collection, you can easily iterate over them to draw each one. Indeed, as you'll see, the *Paint* method consists only of initializing a local counter variable and then iterating over the collection using *ForEach*:

```
procedure TPenPic.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var PenCount: Integer;

  procedure ShowPen(P: PPen); far;
  var
    MemDC: HDC;
    TheBitmap: HBitmap;
  begin
    MemDC := CreateCompatibleDC(PaintDC);
    Inc(PenCount);
    if PenCount = CurrentPen then
      TheBitmap := DownPic
    else TheBitmap := UpPic;
    SelectObject(MemDC, TheBitmap);
    BitBlt(PaintDC, 0, PenCount * 40, 128, 40, MemDC, 0, 0, SrcCopy);

    P^.Select(PaintDC);
```

```
    MoveTo(PaintDC, 15, PenCount * 40 + 20);
    LineTo(PaintDC, 115, PenCount * 40 + 20);
    P^.Delete;
    DeleteDC(MemDC);
  end;

begin
  PenCount := -1;
  PenSet^.ForEach(@ShowPen);
end;
```

The interesting part really comes not so much in *Paint*, but in the nested procedure *ShowPen*, which gets called for each pen in the palette. *ShowPen* is really made up of two distinct parts. The first paints a bitmap as a background, and the second (which will seem very familiar by now) uses the pen object to draw a sample line over the bitmap.

Painting the bitmap takes three steps: creating a memory device context, selecting the bitmap image into the device context, and copying the image into the display context for the screen.

As you saw in Step 8 (page 59), there are different device contexts for various kinds of devices. Windows allows you to create a memory device context for manipulating bitmaps. In fact, you can only select bitmaps into memory device contexts, although those can then be copied to other device contexts.

Painting bitmaps is covered in detail in Chapter 17, "The Graphics Device Interface."

The *CreateMemoryDC* method creates a blank memory device context compatible with the current *PaintDC*. Then, depending on whether this particular pen is the selected pen, *ShowPen* selects either the *UpPic* or the *DownPic* bitmap into the device context. Note that a memory device context treats a bitmap like any other drawing tool. Finally, the *BitBlt* function copies the specified part of the memory device context into *PaintDC*. Note that the memory device context must be disposed of.

Once the bitmapped background is in place, *ShowPen* uses *MoveTo* and *LineTo* just as you did when drawing directly on the application's main window.

Selecting pens with the mouse

Finally, *TPenPic* provides a response to mouse clicks within its painted area. Notice that although the size of the palette object never changes, it only gets mouse click messages in the area that is actually shown on screen. The palette is clipped by the frame of the palette window, which means you can only click on pens that are actually in the palette.

Because each element of the palette is the same size, *WMLButtonDown* can simply divide the y-coordinate of the mouse click (which arrives in *LParamHi*) by 40, the size of each bitmap, to determine which bitmap was clicked. It then makes the clicked bitmap the selected pen and sets the common pen for drawing to be a copy of the selected palette pen. Because there is now a pen selected, it enables the Del Pen button in the palette window, then invalidates the palette to make sure it is redrawn to reflect the new selection.

The code for *WMLButtonDown* follows:

This completes STEP12B.PAS.

```
procedure TPenPic.WMLButtonDown(var Msg: TMessage);
begin
  CurrentPen := Msg.LParamHi div 40;
  if CurrentPen <> nil then Dispose(CurrentPen, Done);
  with PPen(PenSet^.At(CurrentPen))^ do
    CurrentPen := New(PPen, Init(Style, Width, Color));
  PPenPalette(Parent)^.DelBtn^.Enable;
  InvalidateRect(HWindow, nil, False);
end;
```

Where to now?

There are many additions and changes you could make to *Steps* to make it more useful. This section will suggest some changes and some approaches you might use to implement them. A version of *Steps* that incorporates these changes is in the file GRAFFITI.PAS.

Graffiti contains these changes:

- Multiple document interface (MDI)
- Smoother painting
- Undo
- Improved palette window behavior
- Scrolling

Multiple document interface

TStepWindow can very easily work as an MDI child window. In fact, with only slight changes, *Graffiti* uses the same windows for its child windows that *Steps* uses for its main window. Since it's the *TStepWindow* object that owns the pen palette window, each separate drawing can have its own distinct set of pens. The

Multiple Document Interface is explained in Chapter 14, "MDI objects."

Smoother painting

For complex drawings with many long lines, it can take a long time for the picture to redraw itself. *Graffiti* takes advantage of one of the GDI drawing functions called *PolyLine* to draw all the segments of each line at once, rather than one at a time. The effect is faster, smoother painting of the window.

Undo

Because the drawing consists of a collection of lines, it's easy to use the collection's methods to erase the last line drawn. *Graffiti* binds the Edit | Undo menu item to the line collection's *AtDelete* method, removing the last line in the collection, which is also the line most recently drawn. By repeatedly deleting the last line in the collection, you can effectively undo the entire drawing.

Although *Graffiti* doesn't do this, you could also add a Redo function by storing the deleted lines in another collection, and moving them one at a time to the drawing's line collection, or perhaps even to a different drawing.

Palette behavior

You have probably also noticed that as you click between the palette and the main window, the window clicked becomes active (with an active frame), and the other window becomes inactive. If you move between the two windows often (as you might when using a palette), this can be very annoying. To prevent this, you trap the *wm_NCActivate* messages to the windows, and when the message's *WParam* is zero (trying to deactivate the frame), you can change it to one (activating the frame):

```
procedure TPenPalette.WMNCActivate(var Msg: TMessage);
begin
  if Msg.WParam = 0 then Msg.WParam := 1;
  DefWndProc(Msg);
end;
```

Calling *DefWndProc* makes sure the message is processed as usual, but now you've assured the palette's frame is never deactivated. You can add the same trap to *TStepWindow*, as well.

Scrolling

Finally, since each of the MDI child windows tends to be rather small, *Graffiti* adds the ability to scroll the drawing automatically as you draw. In *Steps*, when the pen goes outside the boundaries of the window, the line keeps drawing, even though you can no longer see it. *Graffiti* scrolls the drawing so you can see the part being drawn, and enables you to scroll to other parts of the picture.

PART 2

Using ObjectWindows

C H A P T E R

7

The ObjectWindows hierarchy

ObjectWindows is a comprehensive set of objects that will streamline your development of Microsoft Windows programs in Pascal. This chapter gives an overview of the ObjectWindows object hierarchy. The remaining chapters in this part provide detailed descriptions of different parts of the hierarchy.

In addition to describing the object hierarchy, this chapter outlines the basic principles of programming for the Windows environment, including calling the Windows API.

ObjectWindows conventions

ObjectWindows uses naming conventions for consistency and clarity.

Object names

The names of all object types provided with ObjectWindows start with T. For example, dialog box objects are of type *TDialog*. For every object type definition, there is a corresponding pointer type that starts with a P. For example, a pointer to a *TDialog* object is of type *PDialog*. Throughout the examples in this manual, you will create dynamic object instances, for example, using *PDialog* pointers to allocate *TDialog* objects on the heap.

Method names

Message-response methods are named after the messages they respond to, but without the underscores. For example, a method responding to the *wm_KeyDown* message would be called *WMKeyDown*, and a message responding to the *cm_FileOpen* command would be called *CMFileOpen*.

Overview of the objects

The object hierarchy

ObjectWindows is a hierarchy of object types you can use to handle most of the normal tasks of a Windows application. An outline of the user interface objects in the library appears in Figure 7.1. Figure 7.2 shows the objects in the hierarchy that handle data management and validation.

Figure 7.1: ObjectWindows object type hierarchy

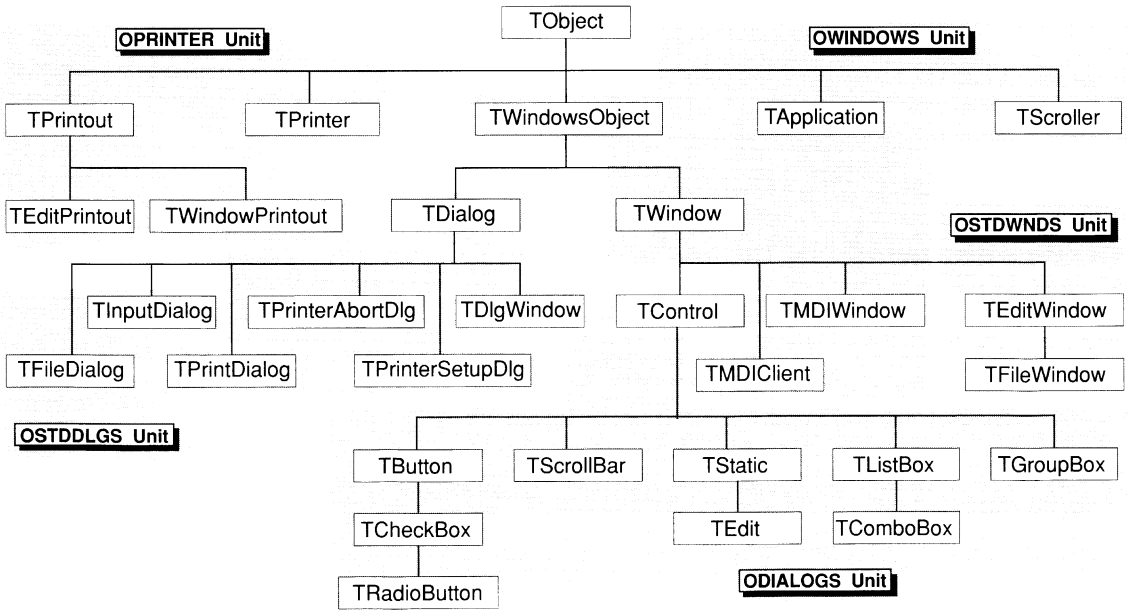
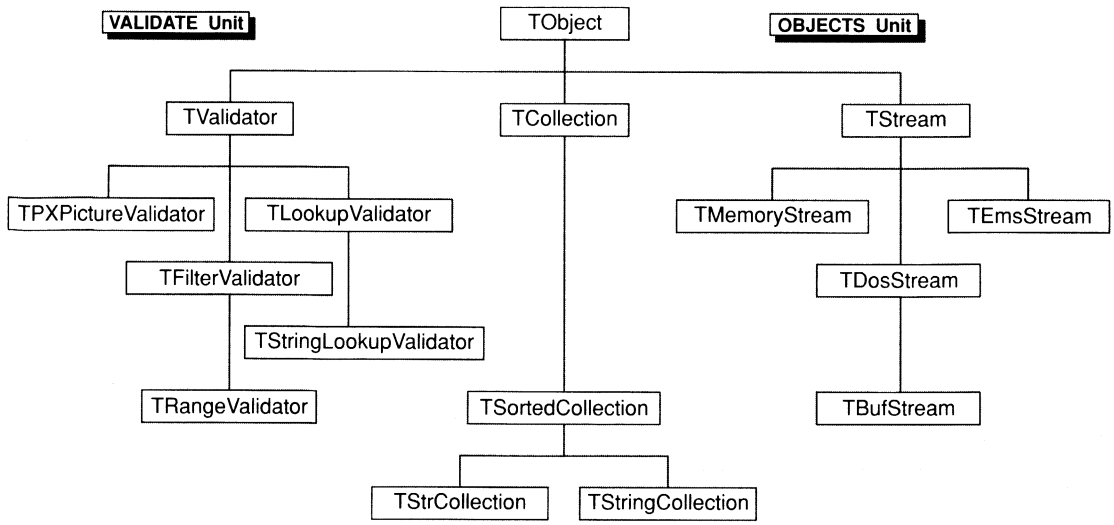


Figure 7.2: Collection, stream, and validator hierarchy



Base object *TObject* is the base object type, the common ancestor of all ObjectWindows objects. It defines a rudimentary constructor and destructor. ObjectWindows streams require that the objects stored on them be descendants of *TObject*.

TApplication This type defines the behavior required of all ObjectWindows applications. Each ObjectWindows application you write will define an application object type derived from *TApplication*. Application objects are described in detail in Chapter 8, "Application objects."

Interface objects The remaining objects in the ObjectWindows hierarchy are classified generally as *interface objects*. They are interface objects in that they represent elements in the Windows user interface, and because they serve as an interface between your application code and the Windows environment. Interface objects are described in detail in Chapter 9, "Interface objects."

Window objects Window objects represent not only the familiar windows of the windowing environment, but also most of the visual elements within that environment, such as controls.

Dialog box objects Dialog box objects provide temporary windows that handle specific input or output functions. Generally they incorporate text and controls such as buttons, list boxes, and scroll bars. Dialog box objects are explained in detail in Chapter 11, "Dialog box objects."

Control objects Within dialog boxes and some windows, controls allow users to enter data and select options. Control objects provide a consistent and simple means of dealing with all the different kinds of controls defined by Windows. Control objects are described in detail in Chapter 12, "Control objects."

MDI objects Windows implements a standard for handling multiple documents within the framework of a single window called the Multiple Document Interface (MDI). ObjectWindows provides a means of setting up and manipulating MDI windows. MDI objects are described in detail in Chapter 14, "MDI objects."

Validator objects ObjectWindows has a full set of data validation objects that can validate user input to edit controls with each keystroke, when the user leaves a field, or when the user completes a window. Data validation is described in detail in Chapter 13, "Data validation."

Printer objects ObjectWindows provides objects that handle printing of documents or printing the contents of a window. Chapter 15, "Printing objects," describes how to use printing objects.

Collection and stream objects The *Objects* unit includes numerous objects that implement flexible data structures called collections and streams that let you store and retrieve objects. Chapter 19, "Collections," describes collections, and Chapter 20, "Streams," covers streams.

ObjectWindows files

ObjectWindows implements the types listed above in compiled units. This section summarizes the content of the supplied units. Only the *OWindows* unit is required in every ObjectWindows application.

This version of ObjectWindows divides different parts of the object hierarchy into separate units. To recompile ObjectWindows

code from earlier versions, you'll generally change all references to the *WObjects* unit to *OWindows*, and add *Objects* and *ODialogs* to programs and units that use collections and streams or dialog boxes, respectively.

This version of ObjectWindows also adds new units for printing, data validation, Borland Windows Custom Controls, and Windows 3.1 support.

Table 7.1 lists the units that make up ObjectWindows and the Windows 3.0 API. The units that support Windows 3.1 extensions are listed in Table 7.2.

Table 7.1
Units for ObjectWindows and the Windows 3.0 API

Unit	Contents
Objects	Base object *TObject*, collections, streams
OWindows	Applications, windows, scrollers, MDI windows
ODialogs	Dialog boxes, dialog windows, controls
OPrinter	Printing, specialized printouts
Validate	Data validators
BWCC	Borland Windows Custom Controls
OStdDlgs	File name dialog boxes, single-line input
OStdWnds	Text editor windows, file editor windows
WinTypes	All types used by Windows 3.0 API routines, including records, styles, messages, and flags
WinProcs	Function and procedure declarations for the Windows 3.0 API

Resource files

The units *OStdDlgs*, *OStdWnds*, and *OPrinter* all have resource files associated with them. The resources for a given unit are in the file named with the unit name and an extension .RES. The resources are automatically included when the corresponding units are used, so any program that uses the *OStdDlgs* unit will automatically have access to the resources in OSTDDLGS.RES.

Windows 3.1 files

In addition to the standard Windows 3.0 API units, you can write programs that take advantage of features added to Windows in version 3.1. Each of the eleven Windows 3.1 DLLs has a corresponding unit:

Table 7.2
Units to access Windows 3.1 features

Unit	Feature
CommDlg	Common dialog boxes
DDEML	Dynamic data exchange messages
Dlgs	Dialog box constants
LZExpand	LZ file expansion
MMSystem	Multimedia extensions
OLE	Object Linking and Embedding

Table 7.2: Units to access Windows 3.1 features (continued)

ShellAPI	Windows shell API
Stress	Strict type checking
ToolHelp	Debugging and other tools
Ver	Versioning
Win31	Windows 3.1 extensions

Interacting with Windows

ObjectWindows insulates you from many of the tedious and confusing parts of the Windows environment, but there will still be times when you need to interact with Windows directly, either because you want to go beyond what is encapsulated by ObjectWindows or because you want to change some behavior defined in ObjectWindows.

There are two ways you can interact with Windows: calling its API functions and receiving messages. This section deals with the API functions. Message handling is covered in Chapter 16, "Windows messages."

Windows API functions

The functionality of Windows lies in its 600 or so functions. Each of these functions has a name, such as *LoadMenu* and *MessageBox*. The only way for a Windows application to manipulate the environment, modify its appearance, or act in response to user input is to call one or a series of Windows functions. Using ObjectWindows, however, you can create windows, display dialog boxes, and manipulate controls, all without calling any Windows functions. How does it work?

ObjectWindows calls API functions

The methods of ObjectWindows call Windows functions. But ObjectWindows isn't duplicating functionality; it's repackaging Windows' list of functions, its application programming interface (API), into an object-oriented library. In addition, ObjectWindows greatly simplifies the task of specifying the numerous parameters required by Windows functions. Often ObjectWindows automatically supplies parameters such as window handles and child-window IDs, which are stored as fields in interface objects.

For example, many Windows functions require a handle to a window to specify which window they are to act upon, and these functions are usually called from the methods of a window object. The object holds the handle of its associated window in its *HWindow* field, so it can pass that as the handle, freeing you from having to specify that item each time. Thus, ObjectWindows's object types serve as an object-oriented layer implemented with calls to the non-object-oriented Windows API.

Access to API functions

In order to directly access any of the Windows functions from your ObjectWindows applications, you must use the *WinProcs* unit. *WinProcs* defines a Pascal procedure or function header for each Windows function, allowing you to call Windows functions just as you would any Pascal routine. See the online Help or the file WINPROCS.PAS for a listing of the function headers.

Windows constants

ws_Popup and the other style constants are defined in the WinTypes unit and are described in Chapter 21, "ObjectWindows reference."

Windows functions require you to pass a variety of *Word*- and *Longint*-type constants as arguments. These constants represent window, dialog box, and control styles, as well as return values and more. Code using these constants is more readable, maintainable, and independent of changes in future versions of Windows than code that uses numbers. For example, it is more informative to define a window with the style *ws_Popup*, rather than $80000000.

Windows data records

Structures used directly by ObjectWindows are listed in Chapter 21, "ObjectWindows reference."

Some functions require more complex data structures, such as those describing fonts (*TLogFont*) or window classes (*TWndClass*). Windows and ObjectWindows define these and other data structures. For a list of the available structures, see the online Help or the file WINTYPES.PAS.

Using ObjectWindows, all the Windows functions are directly available and can be called from within your source code, as long as *WinProcs* appears in the program's **uses** clause. For example, this code calls the Windows function, *MessageBox*, to produce a message box:

```
Reply := MessageBox(HWindow, 'Do you want to save?',
  'File has changed', mb_YesNo or mb_IconQuestion);
```

The value returned by *MessageBox* is an integer, the value of which indicates the action the user took to close the message box. If the user clicked the Yes button, the result is equal to the Windows-defined integer constant *id_Yes*. If the user clicked the No button, the result is *id_No*.

Combining style constants

Windows defines hundreds of style constants, which are listed in Chapter 21, "ObjectWindows reference."

Windows functions that produce interface elements usually require some style parameter of type *Word* or *Longint*. Style-constant identifiers consist of a two-letter mnemonic prefix followed by an underscore and a descriptive name. For example, *ws_Popup* is a window style constants (*ws_* means "window style") for pop-up windows.

Often, these styles must be combined to produce another style. In the *MessageBox* example, you send *mb_YesNo* **or** *mb_IconQuestion* as the style parameter. This style produces a message box with two buttons, Yes and No, and a question mark icon. The **or** bitwise operator actually combines the two constants bit by bit. The resulting style is not one style or the other, but a combination of both.

Keep in mind that some styles are meant to be mutually exclusive. Combining these produces unpredictable, unintended, and probably undesirable results.

Types of Windows functions

Following are the kinds of Windows functions available to your ObjectWindows programs.

Window manager interface functions

These functions handle messages, manipulate windows and dialog boxes, and create system output. This category includes functions for menus, cursors, and the Clipboard.

Graphics device interface (GDI) functions

These functions output text, graphics, and bitmaps on a variety of devices, including the screen and the printer. The functions are not tied to any particular device but are *device independent*.

System services interface functions

These functions handle a wide range of system services, including memory management, interfacing with the operating system, resource management, and communications.

Callback functions

Windows has enumeration functions that permit you to loop over or *enumerate* certain types of elements in the Windows system. For example, you might want to enumerate over the fonts in the system and print a sample of text in each. To use these functions, you must pass a pointer to a function (the *callback function*) for Windows to call during the enumeration process. The specified function will be called directly by Windows as it enumerates over its list of window properties, windows, child windows, fonts, and the like.

Windows functions that require callback functions include *EnumChildWindows*, *EnumClipboardFormats*, *EnumFonts*, *EnumMetaFile*, *EnumObjects*, *EnumProps*, *EnumTaskWindows*, and *EnumWindows*.

The callback function must be a regular function, *not* an object method. The pointer to the function is passed as the first parameter, of type *TFarProc*, in these methods. For example, if you defined the Pascal callback function, *ActOnWindow*, as follows:

```
function ActOnWindow(TheHandle: HWnd; TheValue: Longint): Integer;
  far; export;
```

you could pass it as a callback in a call to the Windows function *EnumWindows*:

```
ReturnVal := EnumWindows(TFarProc(ActOnWindow), ALongint);
```

The callback function must have the same type return value as the Windows function that calls it. The function *ActOnWindow* would take some action on the window specified by the passed handle. The *TheValue* parameter is any value the caller of *EnumWindows* chose to pass.

The {**$K+**} compiler directive handles callback functions for you automatically. If you don't use {**$K+**}, you must pass your callback functions through the *MakeProcInstance* API function to return an address callable by Windows.

C H A P T E R

8

Application objects

When you develop an ObjectWindows application, you first need to define an application object derived from the ObjectWindows type *TApplication*. This object encapsulates the following behavior of an ObjectWindows application:

- Creating and displaying the application's main window
- Initializing the first instance of an application for any tasks that serve all instances of the application, such as creating files for all instances to share
- Initializing every instance of an application, for example, to load an accelerator table
- Processing Windows messages
- Closing the application

Besides defining an application object, you must add to it the ability to construct the main window object. You also have the option to refine the default behavior for initializing instances, closing the application, and processing Windows messages.

The minimum requirements

To be a working Windows application, your ObjectWindows program needs to do only three things in its main **begin..end** block. It must

- Initialize itself
- Handle messages
- Shut down when told to

The default application object handles these tasks by calling three of its methods, *Init*, *Run*, and *Done*. The main block of any ObjectWindows program usually consists of just those three methods. To modify the default behavior of the program, you override the methods.

Finding the application object

Whenever the program is running, ObjectWindows maintains the global variable, *Application*, a pointer to the application object. This pointer allows routines outside the application object to access the object's fields and methods. By default, *Application* is set to *@Self* by the application object's constructor and set to **nil** by the application object's destructor.

The minimal application

Here's a minimal ObjectWindows application:

```
program MinimalApp;
uses OWindows;
var
  MyApp: TApplication;
begin
  MyApp.Init('TestApp');
  MyApp.Run;
  MyApp.Done;
end.
```

MinimalApp is the absolute minimum ObjectWindows application, and it requires no definition of new object types. Typically you'll define new object types for at least the application and the main window.

Init, Run and Done

Since an ObjectWindows program's main block uses the same three statements every time, you need to understand what each of them does (see Figure 8.1).

The Init constructor

The first statement is a call to the application object's *Init* constructor. Calling *Init*

- Constructs the object
- Initializes the object's data fields
- Sets the global variable *Application* to point to the object (*@Self*)
- Performs two kinds of initializations:
  - Calls *InitApplication* if there are no other instances of this application running
  - Calls *InitInstance* always, which sets up the main window by calling *InitMainWindow*

When *Init* has finished, the main window of your application is on the screen. In most cases, you will only redefine *InitMainWindow*, as described on page 104.

The Run method

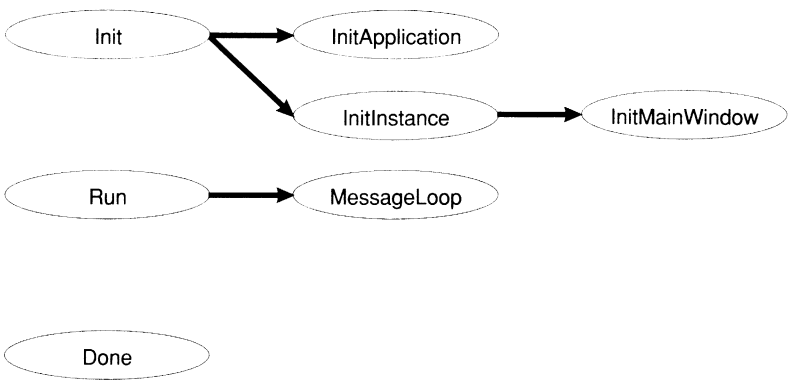
The second statement calls the application's *Run* method, which sets the application in motion by calling *MessageLoop*. *MessageLoop* processes messages from Windows, the instructions that direct an application's activity. *MessageLoop* is, as its name implies, a loop that continues running until the application closes.

MessageLoop calls several methods that process the particular incoming messages, as explained on page 107.

The Done destructor

Done is the destructor for application objects; it frees the application object's memory before the application terminates.

Figure 8.1
Method calls that control an application's flow



Initializing applications

The flow of method calls that initialize the application object allow you to customize parts of the process by redefining particular methods. The one method you must redefine to have a meaningful application is *InitMainWindow*. You can also redefine *InitInstance* and *InitApplication*.

Initializing the main window

You must define an *InitMainWindow* method that constructs and initializes the main window object and stores it in the application object's *MainWindow* field. Listing 8.1 shows a simple application's object declaration and *InitMainWindow* method.

Window objects are described in detail in Chapter 10, "Window objects."

This method creates a new instance of the ObjectWindows *TWindow* type (*PWindow* is a pointer to the type *TWindow*). Normally, your program will define a new window type for its main window, and your *InitMainWindow* will use that new type instead of *TWindow*.

This simple ObjectWindows application incorporates the new *TMyApplication* into the earlier *MinimalApp*. It differs from *MinimalApp* only in that its main window has a caption:

Listing 8.1
A minimal application with a captioned window

```
program TestApp;
uses OWindows;

type
  TMyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil, 'The Main Window'));
end;

var MyApp: TMyApplication;
begin
  MyApp.Init('TestApp');
  MyApp.Run;
  MyApp.Done;
end.
```

TestApp displays a window with the caption 'The Main Window'. You can move and resize this window, minimize it, restore it, or maximize it. Closing the window terminates the application. In

short, *TestApp* is a fully-functional, bare-bones application featuring only the simplest main window.

Showing the main window specially

The initial appearance of the main window is controlled by a variable in the *System* unit, *CmdShow*. *CmdShow* holds one of the *sw_* constants and is passed as a parameter to the Windows API function *ShowWindow* when the application creates its main window.

Using *CmdShow*, you can have your main window appear normally (the default), zoomed to the full screen, minimized to an icon, or hidden entirely. The best place to assign a value to *CmdShow* is the constructor of your main window object, assuring that your chosen value is passed when the screen element is created.

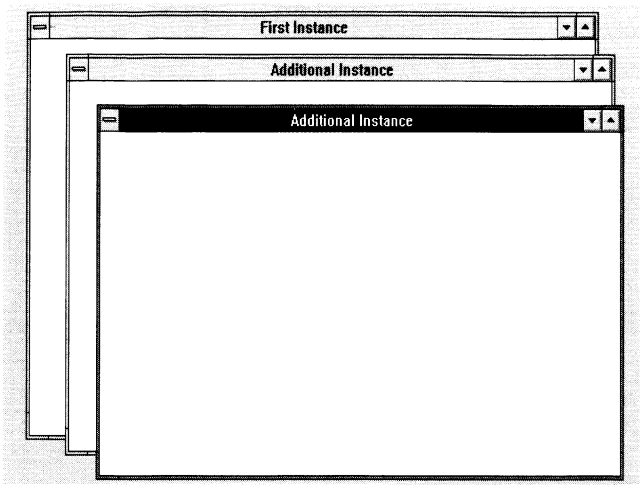
Initializing the first instance

If a user runs your application when no other instance of it is already running, you can define some processing to occur only the first time it is run. This processing is called first instance initialization. If a user starts and terminates your application, starts it again, and so on, each instance would be a first instance because no two instances run at the same time.

If the current instance is a first instance, *Init* calls *InitApplication*. *TApplication* declares a placeholder *InitApplication* method you can redefine to perform special first instance initialization.

For example, you can modify *TestApp* to make the main window's caption reflect whether it is the first instance. To do this, add a Boolean field called *FirstApp* to the application type *TMyApplication*, then define an *InitApplication* method that sets *FirstApp* to *True*. Only the first instance of the application will have a *True* value for *FirstApp*. Finally, modify *InitMainWindow* to check *FirstApp* and display an appropriate caption on the application's main window, as shown in Figure 8.2.

Figure 8.2
Refining application initialization




```
program TestApp;
uses OWindows;

type
  TMyApplication = object(TApplication)
    FirstApp: Boolean;
    procedure InitMainWindow; virtual;
    procedure InitApplication; virtual;
  end;

procedure TMyApplication.InitMainWindow;
begin
  if FirstApp then
    MainWindow := New(PWindow, Init(nil, 'First Instance'))
  else MainWindow := New(PWindow, Init(nil, 'Additional Instance'));
end;

procedure TMyApplication.InitApplication;
begin
  FirstApp := True;
end;

var MyApp: TMyApplication;
begin
  MyApp.Init('TestApp');
  MyApp.Run;
  MyApp.Done;
end.
```

Initializing each instance

A user can run multiple instances of an ObjectWindows application simultaneously. The *InitInstance* method initializes each instance of the application. It should only initialize the application itself, not the application's main window. Initialize the main window in *InitMainWindow*.

InitInstance calls *InitMainWindow* and then creates and shows the main window. You need only redefine *InitInstance* to modify the standard initialization of the application, for example, to load an accelerator table. If you redefine *InitInstance* for your application, be sure it calls the *InitInstance* inherited from *TApplication* first.

Here is an *InitInstance* method that loads an accelerator table before the application is set in motion. 'MyHotKeys' is the resource ID of the accelerator table defined in the resource file.

```
procedure TEditApplication.InitInstance;
begin
  inherited InitInstance;
  HAccTable := LoadAccelerators(HInstance, 'MyHotKeys');
end;
```

You can also use *InitInstance* to register the application instance with an external DLL, such as the Paradox Engine.

Running applications

Your application's *Run* method calls *MessageLoop*, which invokes your program's message loop. While your program runs, the message loop processes incoming Windows messages. ObjectWindows programs inherit a *MessageLoop* method that works automatically. Refine the message loop only for special dialog, accelerator, or MDI handling.

The *MessageLoop* method calls three translation methods to process Windows messages. *ProcessDlgMsg* handles modeless dialog boxes, *ProcessAccels* handles accelerators, and *ProcessMDIAccels* handles accelerators for MDI applications. For applications that do not use accelerators or modeless dialog boxes or are not MDI applications, you might want to streamline the *MessageLoop* somewhat. See the entries for the translation methods under *TApplication* in Chapter 21.

Closing applications

Calling *Done* in your application's main program disposes of the application object. However, before that happens, the program must break out of the message loop. This might happen when the user tries to close the main window of the application. We say it *might* happen because ObjectWindows provides a mechanism to refine an application's closing behavior: You can put conditions on closing.

Modifying closing behavior

All window objects inherit a Boolean *CanClose* method that returns *True* by default, signifying that it's OK to close: That's why *TestApp* closes without hesitation. You can redefine the *CanClose* methods of the application or main window type to change their closing behavior. If any object returns *False* from *CanClose*, the application won't terminate. Normally, you refine the closing behavior of the main window object type. For example, you might want to make sure the application saves files before it terminates.

The CanClose mechanism

The *CanClose* mechanism gives the main window, the application object, and any other windows a chance to either prepare for closing or prevent the closing from taking place. In the end, the application object has to approve the closing of the application. By default, it checks with the main window before approving. The normal closing sequence looks like this:

1. Windows sends a *wm_Close* message to the application's main window.
2. The main window object calls the application object's *CanClose* method.
3. The application object calls the main window object's *CanClose* method.
4. The main window object calls *CanClose* for each of its child windows and returns *True* only if all child windows' *CanClose* methods return *True*.

Modifying CanClose

CanClose methods should rarely return *False*. Instead, they should perform any actions necessary to allow them to return *True*. *CanClose* should only return *False* if unable to do something necessary for orderly shutdown, or if the user indicates a desire to keep the program running.

For example, an editor window's *CanClose* method would probably check to see if the editor text has changed, then prompt the user with a dialog box, asking whether to save the text before closing, and accepting answers of Yes, No, or Cancel. Cancel would indicate that the user doesn't want to close the application yet, so *CanClose* would return *False*. *CanClose* would probably also return *False* if it encounters an error in saving the text, giving the user another chance to save the data before closing.

Unless it redefines *CanClose*, the main window type inherits the *CanClose* of *TWindowsObject*, which returns *True* after calling the *CanClose* methods of its child windows. To modify the closing behavior of your main window, redefine a *CanClose* method for your main window type. It can check for open files, for example, or verify that the user intended to close the window.

C H A P T E R

9

Interface objects

Objects representing windows, dialog boxes, and controls are called interface objects. This chapter discusses the general requirements and behavior of interface objects and their relationship with the actual windows, dialog boxes, and controls that appear on the screen.

The material here applies to all interface objects, for windows, dialog boxes, and controls.

This chapter also explains the relationships between the different interface objects of an application, as well as the mechanism for passing Windows messages.

Why interface objects?

Why do you need interface objects if Windows already has windows, dialog boxes, and controls?

One of the greatest difficulties of Windows programming is that directing visual elements can be awkward and confusing. Sometimes you send a message to a window; other times you call a Windows API function. And the conventions differ for different kinds of elements on the screen.

ObjectWindows alleviates much of this difficulty by providing objects that encapsulate the elements on the screen, insulating you from having to deal directly with Windows and providing a more uniform interface.

What do interface objects do?

The interface object provides methods for creating, initializing, managing and destroying its associated screen element, and has fields that hold data, including its interface element's handle and its parent and child windows. The object's methods manage many of the details of Windows programming for you.

The object/element relationship is a lot like that between an DOS file and a Pascal file variable. With a file, you assign a file variable to represent the physical structure of the actual disk file, then manipulate the file variable. With ObjectWindows, you define an object to represent a physical window, control, or dialog box that is actually managed by the Windows window manager. You work with the object, and it takes care of maintaining the element on the screen.

The generic interface object

ObjectWindows' interface objects all descend from a single abstract object type, *TWindowsObject*. *TWindowsObject* defines behavior common to window, dialog, and control objects, which are refined and specialized in the derived object types *TDialog*, *TWindow*, and *TControl*.

As the common ancestor of all interface objects, *TWindowsObject*'s methods provide uniform ways to:

- Maintain the relationship between objects and screen elements, including creating and destroying objects
- Handle parent-child relationships between interface objects
- Register new Windows classes (See Chapter 10, "Window objects.")

You rarely derive new types directly from *TWindowsObject*, but it serves as the foundation for the object-oriented windowing model. It defines much of the functionality an object inherits when you derive new types from *TWindow* and *TDialog*.

Creating interface objects

Setting up a complete interface object with an associated interface element requires two steps:

- Constructing the object
- Creating the screen element

The first step is calling the constructor *Init*, which constructs the interface object and sets its attributes, such as its style and menu.

Window handles are explained in Chapter 7, "The ObjectWindows hierarchy."

The second step is calling the application object's window creation method, *MakeWindow*, which associates the interface object with a new screen element. This association is maintained by the object's *HWindow* field, a handle to a window.

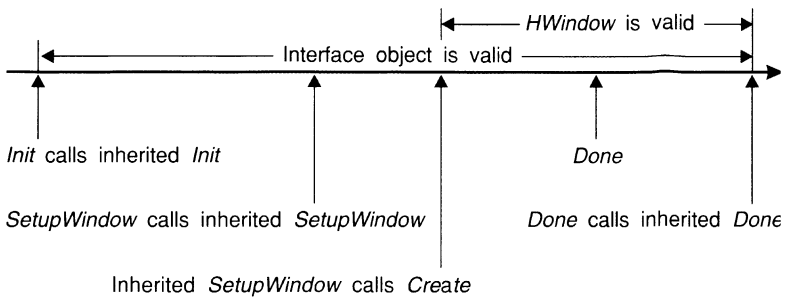
MakeWindow calls the object's *Create* method, which actually tells Windows to create an element on the screen. *Create* also calls *SetupWindow*, which initializes the interface object by, for example, creating child windows.

When is a window handle valid?

Normally under Windows, a newly created interface element receives a *wm_Create* message from Windows, which it responds to by initializing. ObjectWindows interface objects will not receive *wm_Create* messages, so be sure to define a *SetupWindow* method to perform initialization.

If an interface object's initialization requires the screen element's handle (for a call to the Windows API, for example), it must not be called sooner than the *SetupWindow* method. That is, prior to the time *SetupWindow* is called, the interface object's *HWindow* field is *not* valid, and must not be used. If you want to call API functions such as *SetWindowText*, *MoveWindow*, or anything else that requires a window handle, don't call it in the *Init* constructor. Put all such calls in the *SetupWindow* method.

Figure 9.1
When a window has a valid handle



Making things visible

Creating an interface object and its corresponding visual element doesn't necessarily mean that you'll see something on the screen. When the *Create* method has Windows create the screen element, Windows checks to see if the window's style includes *ws_Visible*. If it does, the interface element will be displayed, if not, the element is hidden.

ws_Visible and other window styles are usually set or cleared in the *Init* constructor, in the object's *Attr.Style* field.

At any point after the screen element is created, you can show or hide it by calling the interface object's *Show* method.

Destroying interface objects

As with creating interface objects, getting rid of interface objects is a two-step process. You have to

- *Destroy* the visual interface element
- *Dispose* of the interface object

Destroying the screen element is handled by the interface object's *Destroy* method. *Destroy* does two things. It calls Windows' *DestroyWindow* function to get rid of the screen element and sets the object's *HWindow* field to zero. You can therefore tell if an object is still associated with an element on the screen by examining its handle.

Normally you won't destroy windows yourself. It's handled automatically when the window closes.

You can destroy the screen element without disposing of the corresponding object if you want to create and display it again.

When a user closes a window on the screen, ObjectWindows detects that the screen element has been destroyed, sets the

corresponding object's *HWindow* field to zero, and calls the object's destructor, *Done*.

Linking parent and child objects

In a Windows application, screen elements (windows, dialog boxes, and controls) work together through parent-child links. A parent window controls its child windows, and Windows keeps track of these links. ObjectWindows maintains a parallel set of links between the corresponding interface objects.

A child window is a screen element (it doesn't have to be a *window*) that is managed by another screen element. For example, list boxes are managed by the window or dialog box they appear in. They are displayed only when their parent windows are displayed. In turn, dialog boxes are child windows managed by the windows which spawn them.

When you move or close the parent window, the child windows automatically close, and in some cases, move with it. The ultimate parent of all child interface elements in an application is the main window, although you can have parentless windows and dialogs.

Only dialog boxes and windows, *not controls*, can be parent windows. Any interface element can be a child window.

Child-window lists

The child list is maintained automatically, so you will rarely access it directly.

When you construct a child-window object, you specify its parent as a parameter of the *Init* constructor (see Chapter 10 for an example). A child window object keeps track of its parent through the pointer in its *Parent* field. It also keeps track of *its* child window objects in a linked list, stored in its *ChildList* field. The child window currently pointed to by *ChildList* is the last child window created.

Constructing child windows

As with all interface objects, child window objects get created in two steps, constructing the object and creating the screen element. Child window objects should be constructed by the *Init* constructor of the parent window. For example, a window object descending from *TWindow* that contains a button would do something like this:

```
constructor TMyWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  TheButton := New(PButton, Init(@Self, id_TheButton, 'Button text',
    20, 10, 100, 25, True));
end;
```

Note the use of the *Self* pointer to link the child (*TheButton*) with its parent (an instance of *TMyWindow*). Interface object constructors automatically add the new objects to their parents' child window lists.

Creating child screen elements

Automatic creation can be disabled. See "Disabling automatic creation" on page 116.

Once an interface object's child list is constructed, creating the screen elements for the child windows is automatic. Creating the parent window (through a call to *MakeWindow*) includes a call to the parent's *SetupWindow* method. One of the inherited actions of *SetupWindow* is the calling of *SetupWindow* methods for each of the windows in the child list.

When deriving a new object type, remember to have any initialization of the object take place in *SetupWindow*, after calling its inherited *SetupWindow* method. For example,

```
procedure TMyCheckBox.SetupWindow;
begin
  inherited SetupWindow;      { Make sure defaults are handled first }
   ⋮
                              { Perform initialization of object here }
end;
```

Destroying child windows

Calling a parent window's destructor results in calling the destructors of all its child windows, so your programs need not explicitly call the destructors of child windows. The same is true for the *CanClose* method, which returns *True* only after calling *CanClose* for all its child windows.

Disabling automatic creation

To explicitly exclude a child window from the automatic create-and-show mechanism, call its *DisableAutoCreate* method after constructing it. To explicitly add a child window (such as a dialog box, which would normally be excluded) to the automatic create-

and-show mechanism, call its *EnableAutoCreate* method after constructing it.

Iterating child windows

You may want to write methods that iterate over each of a window's child windows to perform a function. For example, you might want to check all of the child check boxes in a window. In that case, use the *TWindowsObject.ForEach* method, as follows:

```
procedure TMyWindow.CheckAllBoxes;

  procedure CheckTheBox(ABox: PWindowsObject); far;
  begin
    PCheckBox(ABox)^.Check;
  end;

begin
  ForEach(@CheckTheBox);
end;
```

The use of *ForEach* (and the similar *FirstThat* and *LastThat* methods) is similar to the like-named methods of *TCollection*. Although ObjectWindows does not use a collection to manage child windows, the iterator methods work the same way.

Finding a certain child

You might also want to write methods that iterate over a window's list of child windows looking for a particular child. For example, you might want to find the first check box that is checked, in a window with many check box child windows. In that case, use the *TWindowsObject.FirstThat* method as follows:

```
function TMyWindow.GetFirstChecked: PWindowsObject;

  function IsThisOneChecked(ABox: PWindowsObject): Boolean; far;
  begin
    IsThisOneChecked := (ABox^.GetCheck = bf_Checked);
  end;

begin
  GetFirstChecked := FirstThat(@IsThisOneChecked);
end;
```

C H A P T E R

10

Window objects

Window objects are interface objects with special capabilities to make dealing with windows much easier. This chapter explains how to create, display, and fill an application's windows. It includes the following tasks:

- Initializing window objects
- Setting creation attributes
- Creating window screen elements
- Setting registration attributes
- Using specialized windows
- Making windows scroll

What are window objects?

The term "window object" refers to any interface object that represents a window, and in Windows, that means almost everything on the screen. ObjectWindows type *TWindow* serves as a template defining most of the fundamental behavior of the main window and any pop-up windows in an application.

Windows that aren't "windows"

TWindow has three descendant types: *TMDIWindow*, *TControl*, and *TEditWindow*, so all of these are also window objects, even though you might not think of them as "windows." The MDI types are used in ObjectWindows applications that conform to the Windows multiple document interface standard. For an explanation of MDI and these types, see Chapter 14. *TControl* defines controls, such as buttons and list boxes, and is covered in Chapter 12. Most often, you create new window types derived from *TWindow*.

This chapter covers the types *TWindow* and *TEditWindow* and includes an example of registering a new Windows class.

Where to find window objects

Every Windows application has a main window. That "window" might show up as an icon, or it might not show up at all (a "hidden" window), but there is a main window somewhere. ObjectWindows applications are no exception: They must have a main window, represented by a window object.

TestApp is shown in Listing 8.1 on page 104.

The *TestApp* example of Chapter 8, "Application objects," is an example of a bare-minimum ObjectWindows program. An ObjectWindows program's main window is usually an instance of type *TWindow* or a descendant type the program defines. Many applications have other windows that are usually child windows to the main window. These additional windows are also instances of *TWindow* or one of its descendants.

For example, a painting program might define a type *TPaintWindow* for its main window and a *TZoomWindow* type for another window that shows a blown-up view of the painting. In this case, both *TPaintWindow* and *TZoomWindow* types would descend from *TWindow*.

Initializing window objects

Window objects represent window elements, connected through a handle stored in the *HWindow* field inherited from *TWindowsObject*. Since a window object has two parts, creating

one takes two steps: initializing the object and creating the visual element, in that order.

Window initialization is the process of creating the Object-Windows window object by calling the constructor *Init*.

```
Window1 := New(PWindow, Init(nil, 'Title of Window 1'));
Window2 := New(PNewWindowType, Init(nil, 'Title of Window 2');
```

Init creates the new window object and sets the *Title* field of *Attr* to the passed *PChar* argument. The first argument in the *Init* call is the window's parent window object. It is **nil** if the window is the main (parentless) window.

Setting creation attributes

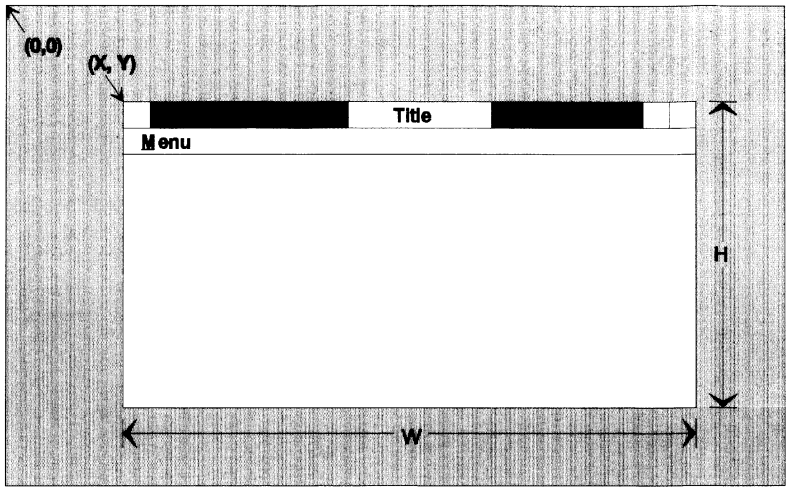
A typical Windows application has many different types of windows: overlapped or pop-up, bordered, scrollable, and captioned, to name a few. These style attributes, as well as a window's title and menu, are set during a window object's initialization and used during a window element's creation.

A window object's creation attributes, such as style, title and menu, are stored in the object's *Attr* field, a record of type *TWindowAttr*. *TWindowAttr* holds the following fields:

Table 10.1
Window creation attributes

Field	Type	Usage
Title	*PChar*	The title, or caption, string
Style	*Longint*	The combined style constant
Menu	*HMenu*	A handle to a menu resource
X	*Integer*	The horizontal screen coordinate of the window's upper left corner
Y	*Integer*	The vertical screen coordinate of the window's upper left corner
W	*Integer*	The window's initial width in screen coordinates
H	*Integer*	The window's initial height in screen coordinates

Figure 10.1
A window's attributes



Default window attributes

As a default, *TWindow.Init* sets *Attr.Style* to *ws_Visible*. If the window is the application's main window, *Style* is *ws_OverlappedWindow* **or** *ws_Visible*.

Menu is set to zero by default, meaning there is no menu defined.

X, *Y*, *W*, and *H* are all set to *cw_UseDefault*, resulting in a reasonably-sized overlapped window. When creating a window that is not the main window, you usually set the *X*, *Y*, *W*, and *H* values yourself.

Overriding default attributes

When you derive new window types from *TWindow*, you usually define a new *Init* constructor, especially if you want a nondefault creation attribute. If you don't want to redefine *Init*, you can still reset a window object's attributes by directly manipulating that object's *Attr* field right after calling *Init*.

If you do redefine *Init*, make sure that the first thing it does is call the inherited *TWindow.Init*, which sets the default attributes. Then you can reset any of the attributes you choose. For example, a typical window type might define an *Init* that sets the *Menu* attribute:

```
constructor TWindowType.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
```

```
  Attr.Menu := LoadMenu(HInstance, 'TheMenu');
  AChildWindow := New(PChildWindowType, Init(@Self, 'Child Title'));
  List1 := New(PListBox, Init(@Self, id_ListBox, 201, 20, 20, 180,
    80));
    :
end;
```

Child-window attributes

TWindowType's constructor is responsible for constructing its child-window objects, such as pop-up windows and list boxes. In turn, a child-window type can set its attributes in its own *Init* constructor:

```
constructor TChildWindowType.Init(AParent: PWindowsObject; ATitle:
  PChar);
begin
  inherited Init(AParent, ATitle);
  with Attr do
  begin
    Style := Style or ws_PopupWindow or ws_Caption;
    X := 100; Y := 100; W := 300; H := 300;
  end;
end;
```

As an alternative, if you choose not to define a descendant window type, you can construct the window object first and then reset its attributes, all from within the parent window's *Init* constructor:

```
constructor TWindowType.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, 'TheMenu');
  AChildWindow := New(PChildWindowType, Init(@Self, 'Child Title'));
  with AChildWindow^.Attr do
  begin
    Style := Style or ws_PopupWindow or ws_Caption;
    X := 100; Y := 100; W := 300; H := 300;
  end;
    :
end;
```

Creating window elements

Once you have constructed a window object, you need to tell Windows to create the screen elements associated with the object.

This is done by calling the application object's *MakeWindow* method, passing a pointer to the window object as a parameter:

```
if Application^.MakeWindow(AWindow) <> nil then
  { creation successful }
else { creation unsuccessful }
```

MakeWindow calls two important methods: *ValidWindow* and *Create*. *ValidWindow* checks whether the window object was constructed successfully by checking its *Status* field. If the constructor failed for any reason, *MakeWindow* returns **nil**. If the constructor was successful, *MakeWindow* goes on to call the window object's *Create* method.

Create is the method that actually tells Windows to create a screen element. Again, if *Create* fails, *MakeWindow* returns **nil**. Otherwise, it returns a pointer to the window object. *Create* also sets the window's *HWindow* field to the handle of the screen element.

Although it is the method that actually creates the window element, you won't normally call *Create* explicitly. The application's main window is automatically created upon application startup by *TApplication.InitInstance*.

Child windows and SetupWindow are described in Chapter 9, "Interface objects."

All other application windows are child windows, either directly or indirectly, of the main window, and child windows are usually either created in the *SetupWindow* method of their parent window objects or dynamically at run time with *MakeWindow*.

In general, parent windows usually call *Init* and *MakeWindow* for their child windows. A window object's attributes are usually set by its methods or by the methods of its parent window object. Since an application's main window has no parent, the application object constructs and creates it upon application startup.

Setting registration attributes

During window object initialization, you can set several attributes of a window, such as its style, location, and menu, by filling the object's *Attr* field. These attributes are used in creating the corresponding window element, so they are called *creation attributes*.

Other attributes, including background color, representative icon, and mouse cursor are more inherent to the window object type and cannot be changed during the operation of the program.

These inherent attributes are called *registration attributes*, because they are set when a window class is registered with Windows.

Windows classes

Associated with every window object type is a list of registration attributes called a *window class*. The list of registration attributes is a lot like the list of creation attributes stored as an *Attr* record in an object field of a window object. However, the registration attributes are stored in a record called *TWndClass* that is defined and maintained by Windows.

The process of associating a window class with a window object type is called *registering* a window class. ObjectWindows automates the registration process. Thus, if you do not want to change any of the window's default characteristics, you need not worry about window class registration.

Table 10.2 shows the fields of a *TWndClass* record and their types:

Table 10.2
Window registration attributes

Characteristic	Field	Type
class style	*style*	*Word*
icon	*hIcon*	*HIcon*
cursor	*hCursor*	*HCursor*
background color	*hbrBackground*	*HBrush*
default menu	*lpszMenuName*	*PChar*

Class style field

This *style* field differs from the window-style (*ws_*) attribute you specify during window initialization because it specifies behaviors inherent to the operation of the window, as opposed to its visual appearance. This field can be filled with one or a combination of class style (*cs_*) constants.

For a complete list of cs_ constants, see Chapter 21, "ObjectWindows reference."

For example, *cs_HRedraw* makes the entire window redraw when its horizontal size changes; *cs_DoubleClk* allows the window to receive double-click messages; *cs_NoClose* inhibits the Close option on the Control menu; and *cs_ParentDC* gives the window its parent's display context.

Icon field

This field holds a handle to an icon that is used to represent a window in its minimized state. Usually, you will define an icon resource to represent the main window of your program.

Cursor field

The *hCursor* field holds a handle to a cursor that is used to represent the mouse pointer when it is positioned over the window.

Background color field

This field specifies the background color of the window. For most applications, the default standard window color, which can be set by the user in the Control Panel, will suffice. However, you can substitute a particular color by setting this field to a handle to a physical brush. Alternatively, you can also set it to any of the Windows color values, such as *color_ActiveCaption*. Always add 1 to any color values.

Default menu field

This field points to a menu resource name that serves as the default menu for this class. For example, if you were to define an *EditWindow* type that always has a standard editing menu, you could specify that menu here. This would eliminate the need to specify the menu in an *Init* method. If this menu resource has an ID of 'MyMenu', you would set this field with

```
AWndClass.lpszMenuName := 'MyMenu';
```

Default registration attributes

Type *TWindow* defines a window class, 'TurboWindow', with a blank icon, an arrow cursor, and standard window color. The default ObjectWindows class, *TurboWindow*, has the following attributes:

- style: *cs_HRedraw* **or** *cs_VRedraw*, redraw after any resize
- icon: *idi_Application*, a blank rectangle
- cursor: *idc_Arrow*, the standard Windows arrow
- background color: *HBrush*(*color_Window* + 1)
- default menu: **nil**

Registering a new class

To change a registration attribute such as the cursor or icon, you need to define a new window class by writing two methods: *GetClassName* and *GetWindowClass*. Any time you change registration attributes, you need to change the class name. If a window class with a given name has already been registered with Windows, other classes with the same class name will not be registered — they get the attributes of the already-registered class.

Changing the class name

GetClassName is a function that returns the name (*PChar*) of the window class. *TWindow.GetClassName* returns 'TurboWindow', the name of the default window class:

```
function TWindow.GetClassName: PChar;
begin
  GetClassName := 'TurboWindow';
end;
```

To define a window object type, called *IBeamWindow*, that uses an I-beam cursor rather than the standard arrow, redefine the inherited *GetClassName* method, as follows:

The class name need not be the same as the object type name. It makes no difference.

```
function IBeamWindow.GetClassName: PChar;
begin
  GetClassName := 'IBeamWindow';
end;
```

Class names should be unique.

Defining new registration attributes

To deviate from the default characteristics, you must fill the fields of a *TWndClass* record with different data in a *GetWindowClass* method.

GetWindowClass takes a *TWndClass* record as a variable argument and fills its fields with new registration attributes. When you define a new *GetWindowClass* method, you should always call the inherited *TWindow.GetWindowClass* first to set the default values, then set the fields you want to change.

For example, the *hCursor* field stores a handle to a cursor resource. For *IBeamWindow*, define the following *GetWindowClass* method:

idc_Beam is a constant representing one of Windows' stock cursors.

```
procedure IBeamWindow.GetWindowClass(var AWndClass: TWndClass);
begin
  inherited GetWindowClass(AWndClass);
  AWndClass.hCursor := LoadCursor(0, idc_IBeam);
end;
```

Besides windows, dialog windows (not dialog boxes) need registered window classes (see Chapter 11). Dialog boxes and controls do not need window classes.

Using specialized windows

ObjectWindows provides two descendants of *TWindow* that are specialized windows for text editing. The *TEditWindow* object type provides a simple text editor that cannot read from or write to a file. The *TFileWindow* type, which descends from *TEditWindow*, provides a text editor that allows reading from and writing to files.

These objects can be used directly as standard components in your applications. You can also derive your own types from them to create specialized editors. Programs or units using edit windows or file windows must include the *OStdWnds* unit in their **uses** clause.

Using edit windows

An edit window is a window with an edit control filling its client area. *TEditWindow.Init* initializes the edit window's *Editor* field to point to an edit control object. *TEditWindow.SetupWindow* sets the dimensions of the edit control to that of the window's client area and creates the edit control's screen element.

The *WMSize* method ensures that the edit control is resized when its window is resized. The *WMSetFocus* method makes sure the edit control gets the input focus when its window receives a *wm_SetFocus* message.

The program *EditWindowTester*, shown in Listing 10.1, uses an edit window to let a user edit text for a simple (nonfunctional) electronic mail application. Figure 10.2 shows this application.

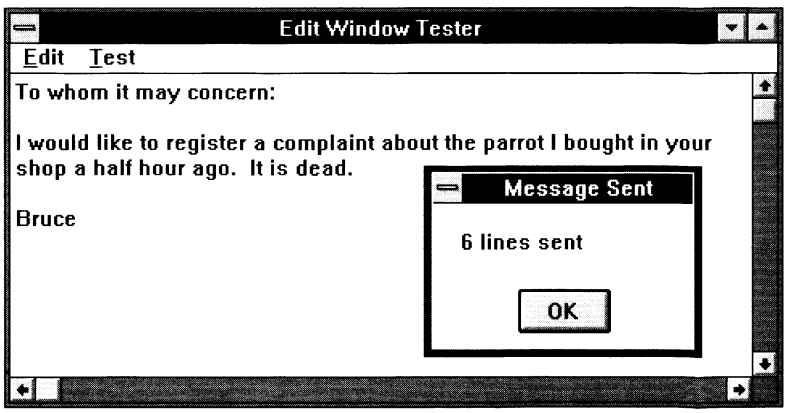


Figure 10.2
An edit window

Listing 10.1
Using an edit window

```
program EditWindowTester;

{$R EWNDTEST.RES}

uses ODialogs, WinTypes, WinProcs, Strings, OStdWnds;

const cm_SendText = 399;

type
  TestApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

  PMyEditWindow = ^MyEditWindow;
  MyEditWindow = object(TEditWindow)
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    procedure CMSendText(var Msg: TMessage); virtual cm_First +
      cm_SendText;
  end;

constructor MyEditWindow.Init(AParent: PWindowsObject;
  ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, MakeIntResource(102));
end;

procedure MyEditWindow.CMSendText(var Msg: TMessage);
var
  Lines: Integer;
  TextString: string[3];
  Text: array[0..20] of Char;
begin
  Lines := Editor^.GetNumLines;
  Str(Lines, TextString);
  StrPCopy(Text, TextString);
  StrCat(Text, ' lines sent');
  MessageBox(HWindow, @Text, 'Message Sent', mb_OK);
end;

procedure TestApplication.InitMainWindow;
begin
  MainWindow := New(PMyEditWindow, Init(nil,
    'Edit Window - Try Typing and Editing'));
end;

var TestApp: TestApplication;
begin
  TestApp.Init('EditWindowTester');
  TestApp.Run;
  TestApp.Done;
end.
```

Using file windows

A file window is an edit window with additional capabilities allowing it to read from and write to files. *TFileWindow.Init* takes the window title as an argument and sets the *FileDialog* field to point to a file dialog object.

TFileWindow has four methods to deal with manipulating files. *Open*, *Save* and *SaveAs* use the *TFileWindow.FileDialog* field (see Chapter 11) to prompt the user for a file name. *New* gives the user a chance to cancel if editing a new file will result in losing changes to the current text. To give the user access to these methods, create your menu with the following menu IDs:

Table 10.3
File window methods and menu IDs

Method	Menu ID to invoke
New	*cm_FileNew*
Open	*cm_FileOpen*
Save	*cm_FileSave*
SaveAs	*cm_FileSaveAs*

You can use file windows as simple standalone text editors without modification. However, sometimes you will want to derive your own types from *TFileWindow* to provide additional functionality. You might want to provide a search facility, for example. Remember, you still have access to the *TFileWindow.Editor* edit control.

Making windows scroll

In most cases, users manipulate scroll bars on the edge of the window's client area to scroll the current view. Unlike standalone scroll bar controls, window scroll bars are part of the window itself.

ObjectWindows handles window scrolling by giving each window object a *Scroller* field which can point to a *TScroller* object. *TScroller* provides an automated way to scroll the text and graphics in windows. In addition, *TScroller* can scroll windows when the user drags the mouse outside of a window's client area; this is *auto-scrolling*, and it works for windows that don't even have scroll bars.

What's a scroller?

TScroller holds values that determine how much of a window is to be scrolled. These values are stored in the *TScroller* fields *XUnit*, *YUnit*, *XLine*, *YLine*, *XPage*, *YPage*, *XRange*, and *YRange*. Fields that start with X represent horizontal values, while those that start with Y represent vertical ones.

Scrolling units

A *scrolling unit* determines the granularity of scrolling. It is the smallest number of device units (usually, pixels in a window, but it depends on the present mapping mode) you can scroll in either the horizontal or vertical direction. The unit is usually based on the kind of information you display.

For example, if you are displaying text that has a character width of 8 pixels and a height of 15, then useful values for *XUnit* and *YUnit* would be 8 and 15, respectively.

Lines, pages, and range

The other scroller attributes, line, page, and range, are expressed in terms of scrolling units. *Line* and *Page* values are the number of units to scroll in response to a user's scrolling request. A request in the form of clicking either of the arrows at the end of a scroll bar scrolls the window "a line's worth," the number of units stored in the line fields. A click in the scroll bar itself (but not on the scroll button, or "thumb") scrolls "a page's worth." Finally, the range attributes represent the total number of units that can be scrolled, usually based on the size of the window's universe, such as the document being edited.

A typical scroller

As an example, consider a text-editing window. If you want to display a text file that has 400 lines of text with a limit of 80 characters per line and 50 lines per page, you might choose these values:

Table 10.4
Typical editing window field settings

Field	Value	Meaning
XUnit	8	character width
YUnit	15	character height
XLine, YLine	1	1 unit per Line
XPage	40	40 characters per horizontal page
YPage	50	50 lines per vertical page
XRange	80	maximum horizontal range
YRange	400	maximum vertical range

A *TScroller* object with these values allows scrolling of one line or page at a time. The entire file can be viewed by using the scroll bars or auto-scrolling.

Default line and page values

The default line values are 1, so it's not necessary to set these unless something else is desired. There is also a default scheme for setting the page units, based on the current size of the window, so that scrolling a "page" will actually scroll the current client area's height or width, depending on the scrolling direction. It isn't necessary to set the page values unless you want to override this mechanism.

Giving your window a scroller

To give your window a scroller, construct a *TScroller* object in your window object's constructor and assign it to the *Scroller* field. You have to set the initial size of the units and the range, but you can change them later.

Window scroll bars are not required for use of a scroller, as in auto-scrolling, but many scrollable windows have them. To add scroll bars to a window, add *ws_VScroll*, *ws_HScroll*, or both, to the window's *Attr.Style* field.

Here is a constructor for the text editing window example:

```
constructor TTextWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  Attr.Style := Attr.Style or ws_VScroll or ws_HScroll;
  Scroller := New(PScroller, Init(@Self, 8, 15, 80, 400));
end;
```

TScroller.Init takes, as arguments, the scrollable window and the initial values for the fields *XUnit*, *YUnit*, *XRange* and *YRange*, respectively. The line and page attributes get default values.

Once this window is displayed, the contents of its client area can be scrolled vertically or horizontally by using the scroll bars or by auto-scrolling. The window's *Paint* method simply draws the graphical information without needing to know that scrolling may take place. Of course, it is not efficient to draw a large picture when only a portion of it is being displayed. The *Paint* method can be optimized to draw only the part of the picture being displayed, as described at the end of this section.

A scrolling example

Scroll is a complete application that draws a graphics design that scrolls. The program, shown in Listing 10.2, draws a series of rectangles that increase in size, so that the entire picture will not fit in the client area of a window drawn on a regular VGA screen. Using the scroll bars, you can view different parts of the design, or you can auto-scroll the picture by holding the left mouse button down within the client area and then moving it out of the area.

Listing 10.2
An example of a scrolling graphics window, found in the file SCROLAPP.PAS.

```
program Scroll;

uses Strings, WinTypes, WinProcs, OWindows;

type
  TScrollApp = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

  PScrollWindow = ^TScrollWindow;
  TScrollWindow = object(TWindow)
    constructor Init(ATitle: PChar);
    procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
      virtual;
  end;

procedure TScrollApp.InitMainWindow;
begin
  MainWindow := New(PScrollWindow, Init('Boxes'));
end;

constructor TScrollWindow.Init(ATitle: PChar);
begin
  inherited Init(nil, ATitle);
  Attr.Style := Attr.Style or ws_VScroll or ws_HScroll;
  Scroller := New(PScroller, Init(@Self, 8, 15, 80, 60));
end;

procedure TScrollWindow.Paint(PaintDC: HDC; var PaintInfo:
  TPaintStruct);
var X1, Y1, I: Integer;
begin
  for I := 0 to 49 do
  begin
    X1 := 10 + I*8;
    Y1 := 30 + I*5;
    Rectangle(PaintDC, X1, Y1, X1 + X1, X1 + Y1 * 2);
  end;
end;
```

```
var ScrollApp: TScrollApp;

begin
  ScrollApp.Init('ScrollApp');
  ScrollApp.Run;
  ScrollApp.Done;
end.
```

Disabling auto-scrolling

TScroller objects can auto-scroll by default, but setting the *AutoMode* field in *TScroller* to *False* turns this feature off. The owning window could do this in its constructor, after constructing the *TScroller* object:

```
Scroller := New(PScroller, Init(@Self, 8, 15, 80, 60));
Scroller^.AutoMode := False;
```

If *AutoMode* is *False*, only the scroll bars can cause scrolling.

One useful feature of auto-scrolling is the fact that the farther you move the mouse out of the window's client area, the faster the window scrolls. Depending on how far out the mouse is, the window scrolls in increments as small as the line value and as large as the page value.

Tracking scroll bars

In addition to auto-scrolling, the program in Listing 10.2 "tracks" the scrolling requests made by moving the scrolling button, or "thumb". In other words, the picture moves as the thumb is moved. This feature gives instant feedback, so that the user can move to the desired part of the picture without having to raise the mouse button.

In some cases, however, tracking may be undesirable. For example, if you are displaying a large file of text, tracking may be slowed by the need to repeatedly read the disk and display the desired portion of text for each movement of the thumb. In such a situation, it is better to disable tracking:

```
Scroller^.TrackMode := False;
```

No scrolling will then take place while moving the thumb until the mouse button is released, when the client area will be scrolled just once to show the correct portion of the picture.

Modifying the units and range

The *Scroll* example assumes that the unit and range values are known at the time of *TScroller* construction. In many cases, this information is not known or can change, such as when the size of the information to be displayed changes. In this case, it may be necessary to set or change the range values (and perhaps the units) at a later time. If you don't know these values at construction, you can pass zero values in the *TScroller* constructor.

Changing the range

The *SetRange* method takes two integer arguments, the number of horizontal and vertical units that determine the total scrolling range. *SetRange* should be used whenever the size of the picture changes. For exampie, when getting ready to display a picture 100 units wide and 300 units high, this command will properly set the scroll range:

```
Scroller^.SetRange(100, 300);
```

Changing units

If the units are not known when the *TScroller* object is initialized, their values must be set before scrolling can take place. For example, they could be set in the window's *SetupWindow* method:

```
procedure ScrollWindow.SetupWindow;
begin
  inherited SetupWindow;
  Scroller^.XUnit := 10;
  Scroller^.YUnit := 20;
end;
```

Modifying the scrolling position

Windows can force scrolling with the *ScrollTo* and *ScrollBy* methods. Each of these takes two integer arguments in terms of horizontal and vertical scrolling units. For example, if it's necessary to reset the scroll position to the upper left corner of the picture, use *ScrollTo*:

```
Scroller^.ScrollTo(0, 0);
```

As another example, if the picture is 400 units long in the vertical direction, the scroll position can be set to the middle of the picture in this way:

```
Scroller^.ScrollTo(0, 200);
```

The *ScrollBy* method moves the scroll position by a number of units up, down, left, or right. Negative values move the scroll position toward the origin, or the top-left corner, and positive values move right and down. If you want to scroll right 10 units and down 20 units, this command will do it:

```
Scroller^.ScrollBy(10, 20);
```

Setting the page size

By default, the page size (*XPage* and *YPage*) is set to the size of window's client area. The scroller is notified when its owning window's size changes. The owning window's *WMSize* method calls its scroller's *SetPageSize* method, which sets its *XPage* and *YPage* fields based on the current size of the window's client area and the values of *XUnit* and *YUnit*.

To override this mechanism and set the page size directly, you override your window object's inherited *WMSize* method to *not* call *SetPageSize*:

```
procedure TTestWindow.WMSize(var Msg: TMessage);
begin
  DefWndProc(Msg);
end;
```

Then you can set *XPage* and *YPage* directly from the window's constructor (or in a *TScroller* descendant's constructor):

```
constructor TScrollWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
begin
  inherited Init(AParent, ATitle);
  Attr.Style := Attr.Style or ws_VScroll or ws_HScroll;
  Scroller := New(PScroller, Init(@Self, 8, 15, 80, 400));
  Scroller^.XPage := 40;
  Scroller^.YPage := 100;
end;
```

Optimizing scroller painting

The *Scroll* example application draws 50 rectangles but doesn't attempt to determine if all the rectangles are actually visible in the window's client area. This might result in wasted effort spent drawing nonvisible graphics. The *TScroller.IsVisibleRect* function can be used in a window's *Paint* method to optimize window painting.

The *ScrollWindow.Paint* method below uses *IsVisibleRect* to determine whether to call the Windows function *Rectangle*. *Rectangle* takes arguments in device units, while *IsVisibleRect* is in terms of scrolling units. For this reason, the values *X1* and *Y1*, the rectangle's origin, and (*X2*-*X1*) and (*Y2*-*Y1*), the rectangle's width and height, must be divided by the respective unit values before calling *IsVisibleRect*:

```
procedure TScrollWindow.Paint(PaintDC: HDC; var PaintInfo:
  TPaintStruct);
var X1, Y1, X2, Y2, I: Integer;
begin
  for I := 0 to 49 do
  begin
    X1 := 10 + I*8;
    Y1 := 30 + I*5;
    X2 := X1 + X1;
    Y2 := X1 + Y1 * 2;
    if Scroller^.IsVisibleRect(X1 div 8, Y1 div 15, (X2 - X1) div 8,
      (Y2 - Y1) div 15) then
      Rectangle(PaintDC, X1, Y1, X2, Y2);
  end;
end;
```

C H A P T E R

11

Dialog box objects

Dialog box objects are interface objects that encapsulate the behavior of dialog boxes: the child windows that perform a specific task, such as configuring a printer or accepting text entry. The *TDialog* object supports the initialization, creation, and execution of all types of dialog boxes. As with window objects, you can derive dialog box objects from *TDialog* for each dialog box in your application.

ObjectWindows supplies standard objects for two common tasks: text input and file selection. ObjectWindows also supplies a type, *TDlgWindow*, that allows you to create a dialog box that behaves more like a window.

This chapter covers the following topics:

- Using dialog box objects
- Manipulating controls in dialog boxes
- Associating objects with controls
- Creating windows from resources
- Using standard dialog boxes for input and file selection

Using dialog box objects

Using dialog box objects is a lot like using pop-up window objects. Dialog boxes are child windows of a parent window. For simple dialog boxes that appear for only a short time, all of the

dialog box handling can be done by one method of the parent window object. The dialog box object can be constructed, executed, and disposed of all in one method, with no need to store it in a field. For more complex dialog boxes, you might want to store the dialog box object in a field of a window object.

Like pop-up windows and controls, dialog boxes are child windows, and are added to the *ChildList* of their parent windows when constructed.

Using a dialog box object requires the following steps:

- Constructing the object
- Executing or running the dialog box
- Closing the dialog box

Constructing the object

Dialog boxes are designed and specified using a resource description created outside the program code. The dialog box resource describes the appearance and placement of controls, such as buttons, list boxes, edit areas, and text strings. It describes only the appearance of the dialog box and does not deal with behavior: that is the application's responsibility.

Each dialog box resource has an identifier, which can be either an ID number (*Word*) or a string (*PChar*). This identifier enables a dialog box object to specify which resource it uses to define its appearance.

Calling the constructor

To construct a dialog box object, call the *Init* constructor. *Init* takes a pointer to the parent window and a *PChar* representing the dialog resource identifier as its parameters:

```
ADlg := New(PSampleDialog, Init(@Self, 'EMPLOYEEINFO'));
```

If the resource ID is a number, it must be typecast to a *PChar* using *MakeIntResource*:

```
Dlg := New(PSampleDialog, Init(@Self, MakeIntResource(120)));
```

The parent window is nearly always *Self*, since dialog boxes are normally constructed within a method of a window object. Dialog box objects not created by window objects should have *Application^.MainWindow* as their parent, since that is the only window object always present in every ObjectWindows program.

Executing dialog boxes

Executing, or running, a dialog box is analogous to creating and displaying a window. However, since dialog boxes are usually displayed for a shorter time period, some of the steps can be abbreviated. This is dependent on whether the dialog box will be displayed as a modal or modeless dialog box.

Modal and modeless dialog boxes

Most dialog boxes are *modal*. Modal means that while the dialog box is displayed, the user cannot select or use its parent window. The user must use the dialog box and close it before proceeding with the program. A modal dialog box, in effect, freezes the operation of the rest of the program.

A *modeless* dialog box does not freeze the operation of the program. It can be created and displayed in a single step, *MakeWindow*, just as a window object can:

```
Application^.MakeWindow(ADlg);
```

You can retrieve data from a dialog box at any time, as long as the dialog box object still exists. This is most often done in the *OK* method, which is the method called when the user presses the OK button.

Executing modal dialog boxes

In the case of modal dialog boxes, it is probably best to construct, execute, and dispose of the object all in one method of the parent window object, as the examples in this chapter demonstrate. That way, every time the dialog box appears, it is a new object.

Application objects have a modal counterpart of *MakeWindow*, called *ExecDialog*. Like *MakeWindow*, *ExecDialog* checks to make sure the dialog box object passed to it is valid (that the object constructor didn't fail, and that the system hasn't run out of memory), then executes the dialog box, making it modal.

ExecDialog returns an integer value that indicates how the user closed the dialog box. The return value is the ID of the control that the user pressed, such as *id_OK* for an OK button, or *id_Cancel* for a Cancel button. After the dialog box finishes executing, *ExecDialog* disposes of the dialog box object.

Thus, with one method call, *ExecDialog*, you can create, display, and end a dialog box:

```
ADlg := New(PSampleDialog, Init(@Self, 'RESOURCEID'));
ReturnValue := Application^.ExecDialog(ADlg);
if ReturnValue = id_OK then
  { Code to retrieve data and process }
else
  if ReturnValue = id_Cancel then { Cancel was pressed }
```

Running modeless dialog boxes

The case of modeless dialog boxes is much like that of pop-up windows and controls. The main reason you cannot dispose of modeless dialog boxes right after running them is that, unlike their modal counterparts, you can't know when the user will close the dialog box. Thus, it is best to construct a modeless dialog in the constructor of its parent window and store it in a field of the parent.

Unlike window and control objects used as child windows, dialog boxes are not automatically displayed when their parent windows are displayed. In the following code, a parent window constructs a dialog box for later modeless use:

```
constructor TParentWindow.Init(AParent: PWindowsObject;
  ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  ADlg := New(PSampleDialog, Init(@Self, 'EMPLOYEEINFO'));
end;
```

Then, every time you want to display the dialog, create and show it:

```
begin
  Application^.MakeWindow(ADlg)
end;
```

The dialog box object, like any other child window, will be automatically disposed of when its parent window closes.

Managing modeless dialog boxes

Dialog boxes differ from other child windows, such as pop-up windows and controls, in that they are often displayed and destroyed many times during the life of their parent windows but are rarely displayed or destroyed at the same time as their parents. Usually, a program produces a dialog box in response to a menu selection, mouse click, error condition, or other event.

This is not an issue for modal dialog boxes, since they are automatically destructed when they close.

Therefore, you must be sure not to construct new dialog box objects, over and over, without disposing of them. Remember that

all constructed dialog objects are automatically inserted into the child-window lists of their parents.

Ending dialog boxes

Every dialog box must have some way for the user to close it. Most often, it is an OK or Cancel button, or both. Descendants of *TDialog* automatically respond to either button by calling the *EndDlg* method, which ends the dialog. You are free to design new and different ways to end a dialog, as long as they result in a call to *EndDlg*. You may also redefine *OK* and *Cancel* methods to modify the closing behavior.

For example, you can redefine an *Ok* method to copy input data into a buffer outside the dialog box object. If the input is invalid, you can put up a message box or generate a beep. If it is valid, you can call *EndDlg*. The integer value passed in *EndDlg* becomes the return value of the call to *ExecDialog*.

As with window objects, dialog objects call *CanClose* before the dialog box closes. You can override *CanClose* to introduce a closing condition, as in the case of a dialog box that verifies user input. If you override *CanClose*, be sure to call the inherited *CanClose* method, as it handles calling of child window *CanClose* methods.

Manipulating controls

To see how to use control objects in a window (not a dialog box), see Chapter 12.

All but the simplest dialog boxes have (as child windows) a series of controls such as edit controls, list boxes, and buttons. Note that these controls are not control objects, but only control interface elements, with no methods or object fields. This chapter also describes an alternative technique that allows you to associate control objects with the control elements of a dialog using *InitResource*.

There is a two-way communication between a dialog object and its control elements. In one direction, the dialog needs to manipulate its controls, for example, to fill a list box. In the other direction, it needs to process and respond to the control event messages generated, for example, when the user selects an item from a list box.

Talking to a control

Windows defines a set of control messages that are sent from the application back to Windows. For example, list box messages include *lb_GetText*, *lb_GetCurSel*, and *lb_AddString*. Control messages specify the specific control and pass along information in *wParam* and *lParam* arguments. Each control in a dialog resource has an ID number, which you use to specify the control to receive the message. To send a control message, call *TDialog*'s *SendDlgItemMsg* method. For example, this method fills a dialog's list box with a text item by sending the message *lb_AddString*:

```
procedure TestDialog.FillListBox(var Msg: TMessage);
var TextItem: PChar;
begin
  TextItem := 'Item 1';
  SendDlgItemMsg(id_LB1, lb_AddString, 0, Longint(TextItem));
end;
```

where *id_LB1* is a constant equal to a list box's ID.

If you ever need to get the handle to one of a dialog's controls, use the *GetItemHandle* method:

```
ListBox1Handle := GetItemHandle(id_LB1);
```

Responding to controls

In the other direction, when the user clicks a control, such as pressing a button or making a selection in a list box, a special command message called a *parent-notification message* is received by the control's parent dialog box. Define a message-response method in the parent dialog box type based on the child ID of each child control that needs a special response:

```
TTestDialog = object(TDialog)
  procedure IDBN1(var Msg: TMessage); virtual id_First + id_BN1;
  procedure IDLB1(var Msg: TMessage); virtual id_First + id_LB1;
end;
```

In this example, *id_BN1* is the ID of a button control and *id_LB1* is the ID of a list box. Clicking on the button will result in a message being sent to the dialog box. The dialog box object reacts through the dynamic method with an index based on that button's ID, *IDBN1*.

Notification messages are explained in detail in Chapter 16, "Windows messages."

Parent notification messages come with a notification code, an integer constant identifying what action occurred. For example, selecting an item in a list box produces a message with the code *lbn_SelChange*. (*lbn_* stands for List Box Notification.) Clicking a button produces a message with the code *bn_Clicked*. Typing in an edit control produces a message with the code *en_Changed*. There are notification codes for list boxes, combo boxes, edit controls, and buttons. The notification code is passed in the message parameter *Msg.lParamHi*. To intercept a control notification message, write the response method for the dialog type to handle the important notification codes:

```
procedure TTestDialog.IDLB1(var Msg: TMessage);
begin
  case Msg.LParamHi of
    lbn_SelChange: {Handle selection change};
    lbn_DblClk: {Handle selection double-click};
  end;
end;
```

Controls that have associated objects can respond to notifications for themselves. See "Control notifications" in Chapter 16, "Windows messages."

Example of communication

The file DIALTEST.PAS contains a program whose main window brings up a modal dialog box, defined by the dialog box type, *TTestDialog*. This program features a two-way communication between the dialog object and its controls. *TTestDialog*'s two methods, *IDBN1* and *IDLB1* are child-ID-based response methods which are invoked when controls (child windows) are clicked by the user. For example, when the user clicks the dialog's *BN1* ('Fill List Box') button, the method *IDBN1* is invoked. Similarly, when the user clicks the list box, *IDLB1* is invoked. In the other direction, the code of the *IDBN1* method sends the dialog a control message, *lb_AddString*, using the dialog method *SendDlgItemMsg*, in order to fill the list box with text items.

DialTest also shows how to create a new dialog box, by defining a new dialog box type and associating it with a dialog resource in the call to the constructor *Init* in the *TestWindow.RunDialog* method. The complete program, DIALTEST.PAS can be found on your distribution disks.

Associating control objects

Until now, we have had dialog objects respond to control-notification messages using child-ID-based message response methods. However, sometimes it is preferable to have the control itself respond to a message. For example, you might want an edit control that only allows digits to be entered, or a button that changes styles when pressed. This is straightforward for control objects in windows (see Chapter 12). However, to do this for dialog controls which are created with resource files, you have to use a different constructor to construct the object.

When associating, you create a control object to represent a dialog's control element. This control object gives you the flexibility to customize the control's message responses. It also lets you use the set of control object methods described in Chapter 12.

To associate an object with a control, first define a control object. It should be constructed in the dialog's constructor. However, instead of using the *Init* constructor, as shown in Chapter 12, use *InitResource*, which takes the parent window and control ID (from the dialog resource) as parameters. This results in calls to the control object's message-response methods instead of the default processing for the element. You can then write response methods for your control object. To do this, you must define a new object type, derived from a supplied control type.

Note that by using *InitResource* you can also now manipulate the control using the object's fields and methods, OOP-style, instead of having to send messages to the interface element through the Windows API.

Note also that, unlike setting up a window object, which takes two steps (*Init* and *MakeWindow*), associating an object with a control element is a single-step process, because the control element already exists: It's loaded from the dialog resource. You just have to tell *InitResource* which control from the resource you want to associate the object with, using the control's ID.

Using dialog windows

The major difference between dialog boxes and windows is that a dialog has an associated resource that specifies the type and location of its controls. But a window can have controls, too.

One approach to putting controls in a window is to use control objects, as shown in Chapter 12. Another approach is to merge the capabilities of dialog boxes and windows, as is done in the object type *TDlgWindow*, which produces a hybrid object called a *dialog window*. The second approach provides a more convenient way to design and manage many controls in a window. In addition, it offers dialog boxes some of the more flexible features of windows.

TDlgWindow descends from *TDialog* and inherits its methods, such as *Execute*, *Create*, *Ok*, and *EndDlg*. Like dialog boxes, dialog windows have a corresponding dialog box resource. On the other hand, like windows, dialog windows have an associated window class that can specify, among other things, an icon, cursor, and menu. Because it is associated with a window class, a *TDlgWindow* descendant should redefine *GetClassName* and *GetWindowClass* methods. This class name *must* be the same one listed in the dialog resource.

In most cases, you will run dialog windows as you would windows or modeless dialog boxes, using the *Create* and *Show* methods, rather than the *Execute* method.

A good use of dialog windows is as the main window of an application when the main window must hold many complex controls. For example, a calculator program can have a dialog window as a main window, where the calculator's buttons are specified as button controls in a dialog resource. This would allow the main window to also have a menu, icon, and cursor.

Using stock dialog boxes

ObjectWindows provides standard dialog boxes for performing two kinds of common functions. One, the input dialog box, prompts the user for a single input string. The other, the file dialog box, allows the user to specify a file name and directory, either for opening a file or for saving one.

Using input dialog boxes

Using the OStdDlgs unit will automatically include the resources in OSTDDLGS.RES.

Input dialog boxes are simple dialog box objects, defined by type *TInputDialog*, that prompt the user for a single line of text input.

You can run input dialog boxes as either modal or modeless dialog boxes, but you'll usually run them as modal. Input dialog box objects have an input dialog resource associated with them, provided by ObjectWindows in the file OSTDDLGS.RES.

Every time you construct an input dialog, using the *Init* method, you specify the caption, prompt, and default text of the dialog. Here is a call to the constructor, *Init*, of an input dialog object:

```
var SomeText: array[0..79] of Char;
begin
  AnInputDlg.Init(@Self, 'Caption', 'Prompt', SomeText,
    SizeOf(SomeText))
   ⋮
end;
```

In this example, *EditText* is a text buffer that gets filled with the user's input when the user clicks the OK button.

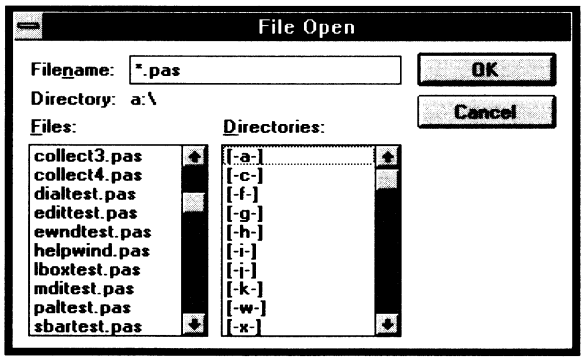
When the user clicks OK or presses *Enter*, the line of text entered in the input dialog is automatically transferred into the character array that passed in the default text. This example constructs and displays an input dialog and retrieves the text:

```
procedure TSampleWindow.Test(var Msg: TMessage);
var
  EditText: array[0..255] of Char;
begin
  EditText := 'Frank Borland';
  if ExecDialog(New(PInputDialog, Init(@Self, 'Data Entry',
    'Enter name:', EditText, SizeOf(EditText)))) = id_OK then
    MessageBox(HWindow, EditText, 'Name is:', mb_OK)
  else MessageBox(HWindow, EditText, 'Name is still:', mb_OK);
end;
```

Using file dialog boxes

File dialog boxes are another type of stock dialog box provided with ObjectWindows in the type *TFileDialog*. Use a file dialog box every time you want to prompt the user for a file name, such as in the File Open, Save, and Save As functions many applications feature. See Figure 11.1.

Figure 11.1
A file dialog box



In most cases, run file dialog boxes as modal dialog boxes (see "Running file dialog boxes" on page 150). Associated with a file dialog box object is a file dialog box resource, provided in Object-Windows in the file OSTDDLGS.RES. Using the unit *OStdDlgs* will automatically include the resource file.

Initializing a file dialog box

TFileDialog defines an *Init* constructor that allows you to specify a file mask and a buffer to retrieve the file name. A file mask, such as '*.TXT', limits the files listed in the file combo box, just as it would in the DOS command, DIR *.TXT. The file name and mask are passed in a record of type *TFileDlgRec*. Here is a sample call to file dialog box's *Init*:

```
var
  FileRec: TFileDlgRec;
  IsOpen: Boolean;
begin
  StrCopy(FileRec.Name, 'TEST1.TXT');
  StrCopy(FileRec.Mask, 'C:\*.TXT');
  IsOpen := True;
  AFileDlg.Init(@Self, FileRec, IsOpen);
   :
end;
```

The last parameter indicates whether the dialog will be an open or save dialog, as described in the next section.

Executing file dialog boxes

There are actually two varieties of the file dialog box: the open file dialog box and the save file dialog box. The difference is the text in the button in the upper right corner of the dialog box. Depending on whether you are asking the user to open or save a file, the button will read Open or Save. When you call *ExecDialog*, you will get the type of dialog box specified by the Boolean parameter in the constructor *IsOpen*. If the file dialog box is constructed with *IsOpen* set to *True*, the dialog box will run as a file open dialog box. If it is constructed with *IsOpen* set to *False*, the file dialog box is a save file dialog box.

One additional feature of ObjectWindows' file dialog box is that it prompts the user if the user tries to save a file with a file name that already exists, as shown in Figure 11.2. Another time you might want to prompt the user is when the user tries to open a new file or clear the current work without having saved it. Since this should happen before the file dialog box is brought up, it is not part of the file dialog box's behavior. The *Steps* example in Part 1 of this book checks the *CanClose* method of its main window before loading a drawing from a file.

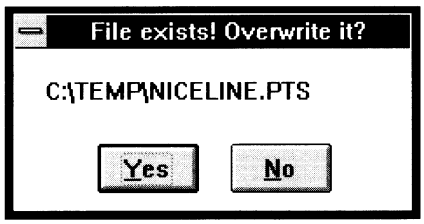


Figure 11.2
Warning the user about overwriting existing files

Here is an example of typical file dialog box use:

```
procedure TMyWindow.OpenSelectedFile;
var FileRec: TFileDlgRec;
begin
  StrCopy(FileRec.Name, 'HEDGEHOG.PAS');
  StrCopy(FileRec.Mask, '*.PAS');
  if ExecDialog(New(PFileDialog,
    Init(@Self, FileRec, True))) = id_OK then
  begin
    Assign(AFile, StrPas(FileRec.Name));
      :
  end;
end;
```

C H A P T E R

12

Control objects

If you need to learn about interface objects, see Chapter 9, "Interface objects."

Windows provides a number of standard user interface devices called *controls*. Controls are special windows with certain expected behavior. ObjectWindows provides interface objects for standard Windows controls so you can use those controls in your applications. Interface objects for controls are called *control objects*.

This chapter covers the following topics:

- Tasks common to all control objects
  - Constructing and destructing control objects
  - Communicating with control objects
- Setting and reading control values
- Using custom controls

In addition, this chapter provides details on using each of the interface objects for Windows' standard controls.

Where do I use control objects?

Dialog boxes and their controls are described in Chapter 11, "Dialog box objects."

Controls are specialized windows that let the user set or select data in a predefined way. The most common place to find controls is in a dialog box. Dialog box controls are defined by the dialog box's resource, so you won't often use objects with them. Modal dialog boxes offer no chance to interact with a control object, so dialog boxes generally only use control objects to set and read the values of controls using *Transfer*. Transferring is the same for

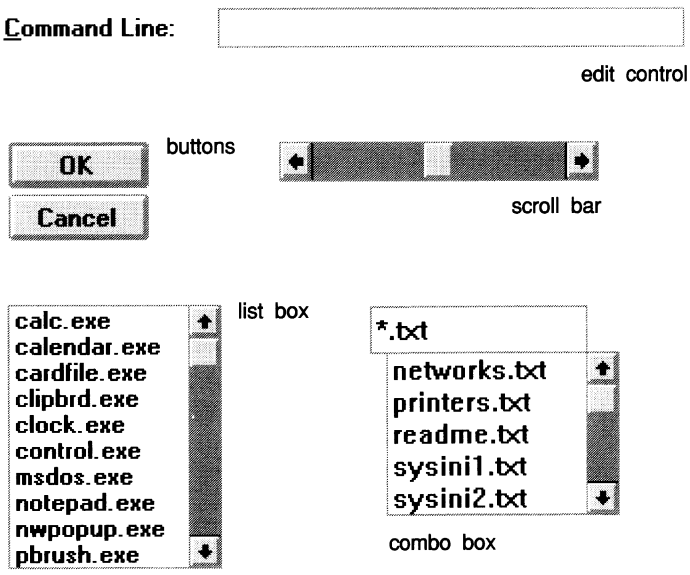
control objects in dialog boxes and windows, and is described in this chapter, starting on page 177.

You will probably also want to use controls in windows, so this chapter focuses on the use of controls outside of dialog boxes. ObjectWindows defines interface object types for all the controls listed in Table 12.1 and shown in Figure 12.1.

Table 12.1
Windows controls supported by ObjectWindows

Control	Object type	Use
list box	*TListBox*	A list of items to choose from
scroll bar	*TScrollBar*	A scroll bar similar to those in scrolling windows and list boxes
push button	*TButton*	A push button with associated text
check box	*TCheckBox*	A button consisting of a box that can be checked or unchecked, with associated text
radio button	*TRadioButton*	A button that can be checked or unchecked, usually used in mutually exclusive groups
group box	*TGroupBox*	A static rectangle with text in the upper left corner
edit control	*TEdit*	A field for the user to type text in
static control	*TStatic*	Visible text the user cannot change
combo box	*TComboBox*	A combined list box and edit control

Figure 12.1
Standard Windows controls



What are control objects?

To Windows, controls are just specialized kinds of windows. In ObjectWindows, type *TControl* descends from type *TWindow*, so much of your use of control objects will be just like using any other window objects. Control objects are similar to window objects in the way they are created and destroyed and in the way they behave as child windows. They differ from other windows, however, in the way they respond to messages. For example, controls' *Paint* methods are disabled. Windows takes care of the painting of its standard controls.

You might find that the ObjectWindows control object types listed in Table 12.1 fill all the needs of your applications. However, there are some cases where you might want to define descendant control types. For example, you might derive a specialized list box type from *TListBox* called *TFontListBox* that holds the names of all the fonts available to your application and automatically displays them when you create an instance of the object.

Type *TControl*, like *TWindowsObject*, is an abstract object type. You can create instances of its descendants, *TListBox*, *TButton*, and the others, but you'll never create an instance of *TControl*.

Note that you'll probably never create a new object type that descends directly from *TControl*, either. *TControl* encapsulates the properties of the standard controls that Windows already knows about. If you create your own controls, you'll most likely derive them from *TWindow*. Creating custom controls is described in this chapter, starting on page 183.

Constructing and destructing control objects

Typically, a parent window object defines a data field for each of its child windows. Regardless of what sort of control object you're using, there are several tasks you perform for each:

- Constructing the control object
- Showing the control
- Destructing the control object

Constructing control objects

Constructing a control is no different from constructing any other child window. Generally the parent window's constructor calls the constructors of all its child windows. In the case of controls, however, you're not just creating a parent-child link, but also establishing a window-control link. This is important because controls communicate with parent windows in special ways (called notifications) in addition to the usual links between parent and child.

Notification is described in Chapter 16, "Windows messages."

To construct and initialize a control object:

- Add a field to the parent window object (optional)
- Call the control object's constructor
- Change any control attributes
- Initialize the control in *SetupWindow*

Calling control object constructors

Where a plain child window object's constructor has only two parameters (its parent window and its title string), a control object has at least six. The list box object has the simplest constructor of any control, with only the six required parameters:

- the parent window object
- the control's ID
- the x-coordinate of the upper left corner
- the y-coordinate of the upper left corner
- the width
- the height

TListBox.Init is declared as follows:

```
constructor TListBox.Init(AParent: PWindowsObject; AnID: Integer;
  X, Y, W, H: Integer);
```

All ObjectWindows control objects (except *TMDIClient*) require at least these six parameters. Most also take a title parameter that provides the text for the control.

Assigning to object fields

Often when you construct a control in a window, you want to keep a pointer to the control in a field of the window object. For example, to add a list box to a window defined by the type *TSampleWindow*, you could give *TSampleWindow* a field called *TheList* and assign the list box to it:

```
constructor TSampleWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
begin
  inherited Init(AParent, ATitle);
  TheList := New(PListBox, Init(@Self, id_LB1, 20, 20, 100, 80));
end;
```

Parent windows automatically maintain a list of their child windows, including controls. However, it's more convenient to manipulate control objects when they have corresponding object fields. Controls that are rarely manipulated, such as static text or group boxes, should probably not have fields dedicated to them.

Constructing a control object without an object field is easy. For example, adding a group box to *TSampleWindow*:

```
constructor TSampleWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
var TempGroupBox: PGroupBox;
begin
  inherited Init(AParent, ATitle);
  TempGroupBox := New(PGroupBox, Init(@Self, id_LB1, 'Group name',
    140, 20, 100, 80));
end;
```

Changing control object attributes

All control objects except *TMDIClient* get the default styles *ws_Child* and *ws_Visible* from calling *TControl.Init*. If you want to change a control's style, you manipulate its *Attr.Style* field, as described for windows in general in Chapter 10. Each control type also has other styles that define its particular properties.

Initializing the control

A control object's screen element is automatically created by the *SetupWindow* method inherited by the parent window object. Make sure that when you derive new window types, you call the inherited *SetupWindow* before any other window initialization.

The controls are also filled and set, if necessary, in the parent window's *SetupWindow*. Here is a typical *SetupWindow* method:

```
procedure TSampleWindow.SetupWindow;
begin
  inherited SetupWindow;                    { creates child controls }
  { Add items to list: }
  TheList^.AddString('Item 1');
  TheList^.AddString('Item 2');
end;
```

Note that control initialization, such as adding strings to list boxes, *must* be performed in *SetupWindow*, not in the constructor. Calling a method such as *TListBox.AddString* causes messages to be sent to the control's screen element. Since the screen element isn't created until the inherited *SetupWindow* is called, attempts to initialize controls before that will fail.

Showing controls

It is not necessary to call *Show* to display controls. As child windows, they are automatically displayed and repainted along with the parent window. However, you can use *Show* to hide or reveal controls on demand.

Disposing of controls

Disposing of controls is the parent window's responsibility. The control's screen element is automatically destroyed along with the parent-window element when the user closes the window or application. The parent window's destructor automatically disposes of its child control objects.

Communicating with controls

Communication between a window object and its control objects is similar in some ways to the communication between a dialog box object and its control elements. Like a dialog box, a window needs a mechanism for manipulating its controls and for responding to control events, such as a list box selection.

Manipulating a window's controls

Dialog boxes manipulate their controls by sending them messages with the *SendDlgItemMsg* method, with a control message constant such as *lb_AddString* as a parameter (see Chapter 11). Control objects greatly simplify this process by using methods, such as *TListBox.AddString*, to directly manipulate the on-screen controls.

When a window's control objects have corresponding object fields, it is simple to call control methods:

```
TheListBox^.AddString('Scotts Valley');
```

Responding to controls

Responding to user interactions with controls is somewhat more complicated than simply calling a control object's methods, because it involves sending and responding to messages. To learn how to respond to control messages, see "Commands, notifications, and control IDs" in Chapter 16, "Windows messages."

Acting like a dialog box

A dialog box with controls allows the user to use the *Tab* key to cycle through all of the dialog box's controls. It also allows the use of the arrow keys to select radio buttons within a group box. To emulate this keyboard interface for windows with controls, call the *TWindowsObject* method *EnableKBHandler* for the window object in its constructor.

Using particular controls

Each different kind of control operates somewhat differently from the others. In this section, you'll find specific information on how to use the objects for each of the standard Windows controls.

Using list box controls

Using a list box is the simplest way to ask the user of a Windows program to pick something from a list. List boxes are encapsulated by the object type *TListBox*. *TListBox* defines methods for four purposes: creating list boxes, modifying the list of items, inquiring about the list of items, and finding out which item the user selected.

Constructing list box objects

TListBox's *Init* constructor takes only the six parameters that all control objects need. Those parameters are the parent window, an ID, and the control's *X*, *Y*, *W*, and *H* dimensions:

```
LB1 := New(PListBox, Init(@Self, id_LB1, 20, 20, 340, 100));
```

TListBox gets the default control style *ws_Child* **or** *ws_Visible*, then adds *lbs_Standard*. *lbs_Standard* is a combination of *lbs_Notify* (to receive notification messages), *ws_VScroll* (to have a vertical scroll

bar), *lbs_Sort* (to sort the list items alphabetically), and *ws_Border* (to have a border). If you want a different list box style, you can modify *TListBox's Attr.Style* field. For example, for a list box that doesn't sort its items, use the following code:

```
LB1 := New(PListBox, Init(@Self, id_LB1, 20, 20, 340, 100));
LB1^.Attr.Style := LB1^.Attr.Style and not lbs_Sort;
```

Modifying list boxes

After you create a list box, you need to fill it with list items, which must be strings. Later, you can add, insert, or remove items or clear the list completely. Table 12.2 summarizes the methods you use to perform these actions.

Table 12.2
Methods for modifying list boxes

To perform this action	Call this method
Add an item	*AddString*
Insert a new item	*InsertString*
Append an item	*InsertString*
Remove an item	*DeleteString*
Delete every item	*ClearList*
Select an item	*SetSelIndex* or *SetSelString*

Querying list boxes

There are five methods you can call to find out information about the list held by a list box object. Table 12.3 summarizes the list box query methods:

Table 12.3
List box query methods

To get this information	Call this method
the number of items in the list	*GetCount*
the item at a particular index	*GetString*
the length of a particular item	*GetStringLen*
the selected item	*GetSelString*
the index of the selected item	*GetSelIndex*

Responding to a list box

The methods for modifying and querying a list box let you set values or find out the status of the control at any given time. In order to know what a user is doing to a list box at run time, however, you have to respond to notification messages from the control.

There are only a few things a user can do with a list box: scroll through the list, click an item, and double-click an item. When one of these actions takes place, Windows sends a *list box notification* message to the list box's parent window. Normally, you'll define a notification-response method in the parent

window object to handle messages for each of the parent's controls.

Every list box notification message contains a list box notification code (a constant with an identifier starting with *lbn_*) in the *lParamHi* field of the *Msg* parameter that specifies the nature of the action. Table 12.4 summarizes the most common *lbn* codes:

Table 12.4
List box notification messages

lParamHi	Action
lbn_SelChange	An item has been selected with a single mouse click.
lbn_DblClk	An item has been selected with a double mouse click.
lbn_SetFocus	The user has given the list box the focus by clicking or double-clicking an item, or by using *Tab*. Precedes *lbn_SelChange*.

Here is a sample parent window method to handle the list box messages:

```
procedure TLBoxWindow.IDLB1(var Msg: TMessage);
var
  ItemText: array[0..9] of Char;
begin
  if Msg.lParamHi = lbn_SelChange then
  begin
    LB1^.GetSelString(@ItemText, 10);
    MessageBox(HWindow, @ItemText, 'You selected:', mb_OK);
  end
  else DefChildProc(Msg);
end;
```

If *Msg.lParamHi* equals *lbn_SelChange*, the user has made a selection, so the response method gets the selected string and shows it in a message box.

Example program: LBoxTest

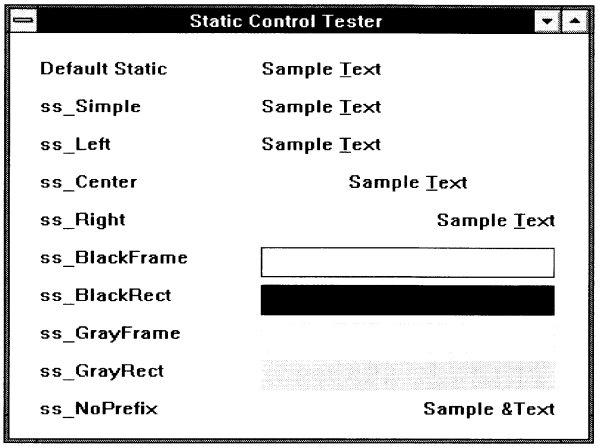
The complete file LBOXTEST.PAS is on your distribution disks.

The program *LBoxTest* is a complete program that creates a window with a list box in it. When the user chooses a list box item, a message box appears with the selected list item displayed. Notice the relationship between the window object and the list box object. The list box is not only a child window of the main window, it is also *owned* by the main window as an object field. One of the main window object's fields is *LB1*, which holds the list box object.

Using static controls

Static controls are usually unchanging units of text or simple graphics that appear on the screen in a window or dialog box. The user does not interact with static controls, although the program can change their text. Figure 12.2 shows the string 'Sample &Text' in a variety of static control styles, with the corresponding Windows style constants.

Figure 12.2
Static control styles



Constructing static controls

Since the user never interacts directly with a static control, the application rarely receives control-notification messages concerning a static control. Therefore, most static controls can be constructed with –1 as the control ID.

TStatic's *Init* constructor is declared as follows:

```
constructor TStatic.Init(Aparent: PWindowsObject; AnID: Integer;
  ATitle: PChar; X, Y, W, H: Integer; ATextLen: Word);
```

In addition to the six usual *Init* parameters for a control object, *TStatic.Init* takes two additional parameters: the text string *ATitle* and the maximum text length *ATextLen*. Because the text must include a terminating null, the number of displayable characters is actually one less than the text length passed to the constructor. A typical call to construct a static control looks like this:

```
Stat1 := New(PStatic, Init(@Self, id_ST1, '&Text', 20, 50, 200,
  24, 6));
```

You will rarely need to access or manipulate a static control object once you have created it, so it's generally unnecessary to assign a field to hold the static control object.

The default static control style is the default control style, *ws_Child* **or** *ws_Visible*, along with *ss_Left* (left justified) style. To change the style, manipulate the *Attr.Style* field. For example, to center the control's text:

```
Stat1^.Attr.Style := Stat1^.Attr.Style and (not ss_Left) or
  ss_Center;
```

One option for static controls is to underline one or more characters in the text string. Insert an '&' in the string immediately preceding the character you want underlined. For example, to underline the T in 'Text', pass the string '&Text' in the *Init* call. If you really want the '&' in the string, use the Windows static style *ss_NoPrefix*, as shown in Figure 12.2.

Modifying static controls

TStatic has two methods for altering the text of a static control. *SetText* sets the static's text to the passed *PChar* argument. *Clear* erases the static's text. You cannot change the text of static controls created with the style *ss_Simple*.

Querying static controls

To retrieve the text held by a static control, use the *GetText* method.

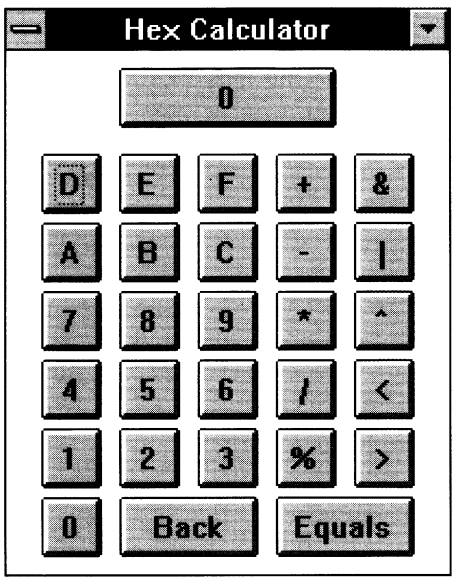
Example program: StatTest

The program *StatTest* creates the static testing application shown in Figure 12.2. Note that the labels, such as 'Default Static' and 'ss_Simple', are static controls, as well as 'Sample Text' and the black and gray boxes. The complete file STATTEST.PAS is on your distribution disks.

Using push button controls

Push buttons (sometimes called "command buttons") perform some task each time the button is pressed. There are two kinds of push buttons, both of type *TButton*, distinguished by the styles *bs_PushButton* and *bs_DefPushButton*. Default push buttons are similar to push buttons but have a bold border, indicating the default user response. Figure 12.3 shows a sample Windows program that uses plain and default push buttons.

Figure 12.3
A Windows program that uses push buttons



Constructing push buttons

In addition to the usual six parameters, *TButton*'s *Init* constructor takes a text string of type *PChar*, *AText*, and a Boolean flag, *IsDefaultButton*, which indicates whether the button should be a default push button or a regular push button. *TButton*'s *Init* is declared as follows:

```
constructor TButton.Init(AParent: PWindowsObject; AnID: Integer;
  AText: PChar; X, Y, W, H: Integer; IsDefault: Boolean);
```

A typical push button constructor for a normal button looks like this:

```
Push1 := New(PButton, Init(@Self, id_Push1, 'Test Button', 38, 48,
  316, 24, False));
```

Responding to push buttons

When the user clicks a push button, the button's parent window receives a notification message. If the parent window object intercepts the message, it can respond to these events by putting up a dialog box, saving a file, or any other program-controlled activity.

To respond to button messages, define a child-ID-based method to handle each button. For example, the following method, *IDBtn1*, handles the response to the user clicking a push button with the ID *id_Btn1*. The only notification code defined by

Windows for push buttons is *bn_Clicked*, so you don't need to check the notification code.

The example program BtnTest on your distribution disks shows the use of push buttons.

```
type
  TTestWindow = object(TWindow)
    Btn1: PButton;
    procedure IDBtn1(var Msg: TMessage); virtual id_First + id_Btn1;
    :
  end;

procedure TestWindow.IDBtn1(var Msg: TMessage);
begin
  MessageBox(HWindow, 'Clicked', 'The Button was:', mb_OK)
end;
```

Using selection boxes

Type *TCheckBox* descends from *TButton* and type *TRadioButton* descends from *TCheckBox*. We will sometimes refer to check boxes and radio buttons collectively as *selection boxes*.

A *check box* generally presents the user with a two-state option. The user can check or uncheck the control, or leave it as is. In a group of check boxes, any or all might be checked. For example, you might use check boxes to pick three fonts for an application to load.

Radio buttons, on the other hand, are used for selecting one of several mutually-exclusive options. For example, you might use radio buttons to pick a font for a particular character.

An important issue concerning a selection box is its state. While displayed on the screen, a selection box is either checked or unchecked. When the user clicks a selection box, it's an event, generating a Windows message. As with other controls, the selection box's parent window usually intercepts and acts on these messages.

However, you might want to derive types from *TCheckBox* or *TRadioButton* to have them perform some action when pressed. If your type defines a method for *nf_First + bn_Clicked*, it should call its ancestor's *BNClicked* response method first and then perform any additional actions desired.

Constructing check boxes and radio buttons

In addition to the usual six parameters, the *Init* constructors for check boxes and radio buttons take a text string and a pointer to a group box object (See "Group boxes" on page 165.) which logically and visually groups the selection boxes. If *AGroup* is **nil**, the selection box is not part of any logical group. The constructors are declared as follows:

```
constructor Init(AParent: PWindowsObject; AnID: Integer; ATitle:
  PChar; X, Y, W, H: Integer; AGroup: PGroupBox);
```

The syntax is identical for both kinds of selection boxes. The constructors differ only in the default styles they assign. A typical use of selection box constructors look like this:

```
GroupBox1 := New(PGroupBox, Init(@Self, id_GB1, 'A Group Box', 38,
  102, 176, 108));
ChBox1 := New(PCheckBox, Init(@Self, id_Check1, 'Check Box Text',
  235, 12, 150, 26, GroupBox1));
```

Check boxes by default have the *bs_AutoCheckBox* style, and radio buttons have the *bs_AutoRadioButton* style. With these styles, only one radio button per group can be checked at any one time. When one button is checked, the others automatically become unchecked.

If you redefine the styles of check box or radio button objects to be "non-auto," you are responsible for managing the checking and unchecking in response to user clicks.

Modifying selection boxes

Modifying (by checking and unchecking) a selection box's state sounds like a job for your program's user, not you. But in some cases, your program needs direct control over a selection box's state. One case where you might want to control a selection box's state is to display options which have previously been selected and saved. *TCheckBox* defines four methods for modifying a check box's state, as shown in Table 12.5.

Table 12.5
Method for modifying selection boxes

To perform this action	Call this method
Check a box	*Check* or *SetCheck(bf_Checked)*
Uncheck a box	*Uncheck* or *SetCheck(bf_Unchecked)*
Toggle a box	*Toggle*

When you use these methods with radio buttons, ObjectWindows ensures that only one radio button per group is checked, as long as the buttons are assigned to a group.

Querying selection boxes

Querying a selection box is one way to find out and respond to its state. Radio buttons and check boxes have two states: checked and unchecked. Use the *GetCheck* method to read the state of a selection box:

```
MyState := Check1^.GetCheck;
```

The example program BtnTest on your distribution disks shows the use of both check boxes and radio buttons.

The return value of *GetCheck* can be compared with the defined constants *bf_Unchecked*, *bf_Checked*, and *bf_Grayed* to determine the state of the box.

Using group boxes

In its simplest form, a group box is a labeled static rectangle that visually groups other controls.

Constructing group boxes

In addition to the usual six control parameters, a group box's *Init* constructor takes a text string to label the group:

```
constructor TGroupBox.Init(AParent: PWindowsObject; AnID: Integer;
  AText: PChar; X, Y, W, H: Integer)
```

A typical use of the group box constructor looks like this:

```
GroupBox1 := New(PGroupBox, Init(@Self, id_GB1, 'A Group Box',
  38, 102, 176, 108));
```

Grouping controls

While a group box visually associates a group of other controls, it can also logically associate a group of selection boxes (check boxes and radio buttons). This logical group performs the automatic unchecking characteristic of the "auto" style selection boxes.

To be added to a group, a selection box specifies a pointer to a group box when it is constructed. For example, to add a group of check boxes to a window, you would include the following code in the window object and its constructor:

```
type
  TSomeWindow = object(TWindow)
    Group: PGroupBox;
    FirstCheck, SecondCheck: PCheckBox;
    constructor Init(AParent: PWindowsObject, ATitle: PChar);
  end;
```

```
constructor TSomeWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
  Group := New(PGroupBox, Init(@Self, id_TheGroup, 'Various boxes',
    10, 10, 100, 50));
  FirstCheck := New(PCheckBox, Init(@Self, id_FirstCheck, 'One',
    15, 20, 90, 10, Group));
  SecondCheck := New(PCheckBox, Init(@Self, id_SecondCheck, 'Two',
    15, 35, 90, 15, Group));
end;
```

Note that the group parameter passed to the selection box is a pointer to the group box object, not the ID of the group control, which is the way the Windows API handles this connection. Using the pointer allows you to construct the objects before the screen elements are created by the parent window's *SetupWindow* method.

Responding to group boxes

When an event occurs that might have changed the group box's selections (for example, when a user presses a button or the program calls *Check*), Windows sends a notification message to the group box's parent window. The parent intercepts the message by using the sum of *id_First* and the group box ID. This lets you define methods for each group instead of for every selection box in the group.

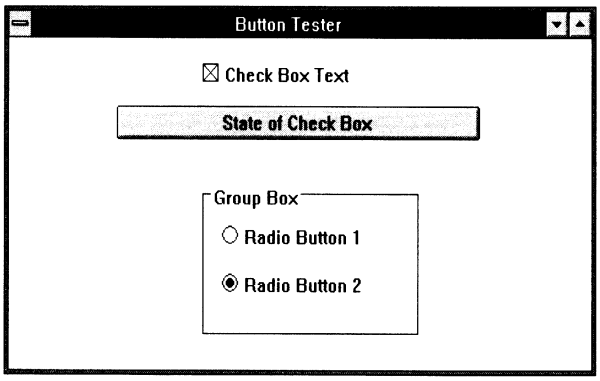
To find out which control in the group was affected, you can read the current status of each control.

Example program: BtnTest

The complete file BTNTEST.PAS is on your distribution disks.

BtnTest is a full program that creates a window with push button, check box, radio button, and group box controls. When the user clicks the controls, the application responds in a variety of ways. See Figure 12.4.

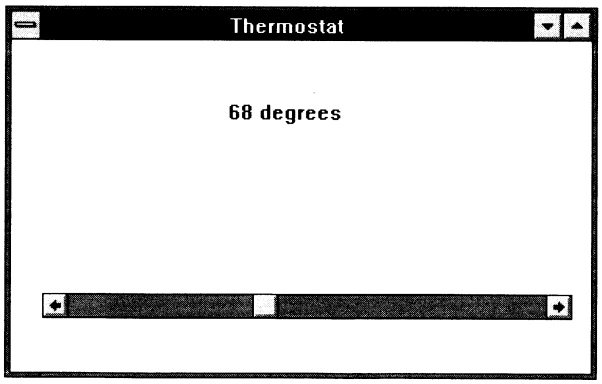
Figure 12.4
A window with various buttons



Using scroll bars

Scroll bars are the primary mechanism for changing the user's view of an application window, a list box, or a combo box. However, you might want a separate scroll bar to perform a specialized task, such as controlling the temperature in a thermostat program or the color in a drawing program. Use *TScrollBar* objects when you need a separate, customizable scroll bar. Figure 12.5 shows a program that uses a *TScrollBar* object.

Figure 12.5
A scroll bar object



Constructing scroll bars

In addition to the usual six control object parameters, a scroll bar's *Init* constructor takes a Boolean flag specifying whether the scroll bar is a horizontal. The following is the declaration of the scroll bar constructor:

```
constructor TScrollBar.Init(AParent: PWindowsObject; AnID: Integer;
  X, Y, W, H: Integer; IsHScrollBar: Boolean);
```

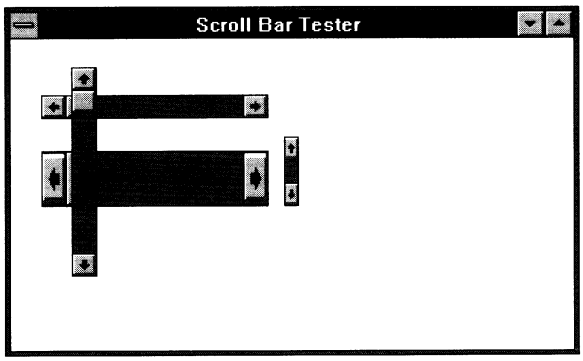
If you specify a width of zero for a vertical scroll bar, Windows assigns it a standard width, like that of a list box's scroll bar. The same holds if you give a height of zero for a horizontal scroll bar. This code

```
ThermScroll := New(PScrollBar, Init(@Self, id_ThermScroll, 20, 170,
  340, 0, True));
```

creates the standard-height horizontal scroll bar shown in Figure 12.5.

Init constructs scroll bars with the styles *ws_Child*, *ws_Visible* and *sbs_Horz* or *sbs_Vert* for horizontal and vertical scroll bars respectively. You can specify additional styles, such as *sbs_TopAlign*, by changing the scroll bar's *Attr.Style* field. Figure 12.6 shows a variety of scroll bar objects.

Figure 12.6
A Window with a variety of scroll bars



Controlling the scroll bar range

One attribute of a scroll bar object initialized at its construction is its *range*. The range of a scroll bar is the set of all possible *thumb* positions. A thumb is the scroll bar's sliding box that the user drags or scrolls. Each position is associated with an integer. The parent window uses this integer, the *position*, to set and query the scroll bar. By default, a scroll bar object's range is 1 to 100.

The thumb's minimum position (topmost in vertical scroll bars and leftmost in horizontal scroll bars) corresponds to the position 1. Accordingly, the thumb's maximum position corresponds to 100. Use the *SetRange* method, described in the section "Modifying Scroll Bars," to set the range differently.

Controlling scroll amounts

A scroll bar object has two other important attributes: its line magnitude and page magnitude. The line magnitude, initialized to 1, is the distance, in *range* units, the thumb moves when the user clicks the scroll bar's arrows. The page magnitude, initialized to 10, is the distance, also in range units, the thumb moves when the user clicks the scrolling area. You can reset these values by directly manipulating a *TScrollBar*'s fields *LineSize* and *PageSize*.

Querying scroll bars

TScrollBar defines two methods for querying a scroll bar: *GetRange* and *GetPosition*. *GetRange* is a procedure that takes two integer variable parameters, setting them to the minimum and maximum thumb positions in the scroll bar's range. Use *GetRange* if you want your program to move the thumb to its minimum or maximum position.

GetPosition is a function that returns the integer position of the thumb. Often your program will get the range and the position and compare the two.

Modifying scroll bars

Modifying scroll bars is mostly a job for the user of your programs, but your program can also modify a scroll bar directly, using the methods summarized in Table 12.6.

Table 12.6
Methods for modifying scroll bars

To perform this action	Call this method
Set scrolling range	*SetRange*
Set thumb position	*SetPosition*
Move thumb position	*DeltaPos*

Be sure to call SetRange only after SetupWindow creates your scroll bar's screen element.

SetRange is a procedure that takes two integer arguments for the lowest and highest positions in the range. You might want to change the default range of 1..100 to better map the actual entity the scroll bar controls. For example, a scroll bar in a thermostat application might have a range of 32 to 120 degrees Fahrenheit:

```
ThermScroll^.SetRange(32, 120);
```

SetPosition is a procedure that takes one integer argument, the new position for the scroll bar's thumb. In the thermostat application discussed earlier, your program could directly set the temperature setting to 78 degrees with

```
ThermScroll^.SetPosition(78);
```

The third method, *DeltaPos*, moves the scroll bar's thumb position up (left) or down (right) by the amount specified by the integer argument. A positive integer moves the thumb down (right). A negative integer moves it up (left). For example, to lower the thermostat's temperature setting by 5 degrees, use

```
ThermScroll^.DeltaPos(-5);
```

Responding to scroll bars

When a user moves a scroll bar, Windows sends a scroll bar notification message to the parent window. If you want your window to respond to scrolling events, respond to the notification messages by defining notification response methods.

Scroll bar notification messages are slightly different than the other control notification messages. They are based on the Windows messages *wm_HScroll* and *wm_VScroll*, rather than on *wm_Command*. The important difference is that the scroll bar notification codes are stored in *Msg.wParam*, rather than in *Msg.lParamHi*.

Common scroll bar notification codes include *sb_LineUp*, *sb_LineDown*, *sb_PageUp*, *sb_PageDown*, *sb_ThumbPosition*, and *sb_ThumbTrack*. Often you can respond to every event by checking the scroll bar's new position and responding accordingly. In this case, you can ignore the notification code. For example,

```
procedure TestWindow.HandleThermScrollMsg(var Msg: TMessage);
var NewPos: Integer;
begin
  NewPos := ThermScroll^.GetPosition;
  { Do some processing based on NewPos. }
end;
```

One common alternative is not to respond to a thumb drag until the user has chosen a new location. In that case, screen out the messages with the *sb_ThumbTrack* code.

```
procedure TestWindow.HandleThermScrollMsg(var Msg: TMessage);
var NewPos: Integer;
begin
  if Msg.wParam <> sb_ThumbTrack then
  begin
    NewPos := ThermScroll^.GetPosition;
    { Do some processing based on NewPos. }
  end;
end;
```

Occasionally you might want the scroll bar object itself to respond to scroll bar notification messages, building a particular response behavior into the scroll bar object. This process is described in "Control notifications" in Chapter 16, "Windows messages."

Example program: SBarTest

The *SBarTest* program creates the thermostat application shown in Figure 12.5. The complete file SBARTEST.PAS is on your distribution disks.

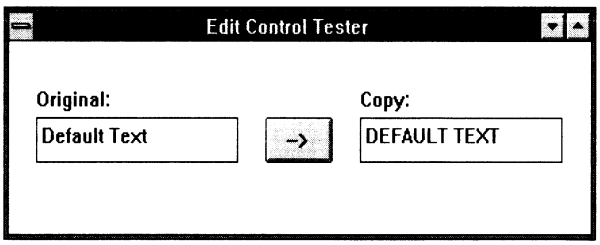
Using edit controls

Edit controls are interactive static controls. They are rectangular areas (boxed or unboxed) on the screen that can be filled with text, modified, and cleared by the user or the application. Edit controls are most useful as fields for data entry screens. They support the following operations:

- User text input
- Dynamic display of text by the application
- Cutting, copying, and pasting to the clipboard
- Multiline editing (good for text editors)

Figure 12.7 shows a window that contains two edit controls.

Figure 12.7
A window with edit controls



Constructing edit controls

The edit control's *Init* constructor is just like that for a static control, taking the usual six parameters plus an initial text string and maximum text length, and adding a Boolean flag, *Multiline*. *TEdit*'s constructor is declared as follows:

```
constructor TEdit.Init(AParent: PWindowsObject; AnID: Integer;
  ATitle: PChar; X, Y, W, H, ATextLen: Integer; MultiLine: Boolean);
```

By default the edit control has the styles *ws_Child* and *ws_Visible*, *ws_TabStop*, *es_Left*, and *es_AutoHScroll*. The text length parameter is actually one greater than the maximum number of characters

allowed in an edit line, as the control must include a terminating null character.

If *Multiline* is *False*, the edit control is a single line edit control and gets the style *ws_Border*. If *Multiline* is *True* the edit control gets the styles *es_MultiLine*, *es_AutoVScroll*, *ws_VScroll*, and *ws_HScroll*. The following are typical edit control constructors, one for a single line control, the other multiline:

```
EC1 := New(PEdit, Init(@Self, id_EC1, 'Default Text', 20, 50, 150,
  30, 40, False));
EC2 := New(PEdit, Init(@Self, id_EC2, '', 20, 20, 200, 150, 40,
  True));
```

Using Clipboard and the Edit menu

You can directly transfer text between an edit control object and the Windows clipboard using method calls. You'll probably want to give users access to these methods through an editing menu.

Edit control objects have built-in responses to menu items such as Edit | Copy and Edit | Undo. *TEdit* defines command-response methods, such as *CMEditCopy* and *CMEditUndo*, which Object-Windows invokes in response to menu choices in the edit control's parent window. *CMEditCopy* calls *Copy* and *CMEditUndo* calls *Undo*.

Table 12.7 shows the clipboard and editing methods and the menu commands that invoke them.

Table 12.7
Edit control commands and the Edit menu

Operation	TEdit method	Menu command
Copy text to Clipboard	*Copy*	*cm_EditCopy*
Cut text to Clipboard	*Cut*	*cm_EditCut*
Paste text from Clipboard	*Paste*	*cm_EditPaste*
Clear entire edit control	*Clear*	*cm_EditClear*
Delete selected text	*DeleteSelection*	*cm_EditDelete*
Undo last edit	*Undo*	*cm_EditUndo*

To add an editing menu to a window that contains edit control objects, define a menu resource for the window using the menu commands listed in Table 12.7. You don't need to write any new methods.

One additional editing method available is the Boolean function, *CanUndo*, which determines if the editor can undo the last change.

Querying edit controls

Often, you want to query an edit control to validate its text entry, store the entry for later use, or copy the entry to another control. *TEdit* supports a number of querying methods. Many of the edit control query and modification methods return, or require you to specify, a line number or a character's position in a line. All of these indexes start at zero. In other words, the first line is line zero and the first character in any line is character zero. The most important query methods are *GetText*, *GetLine*, *NumLines*, and *LineLength*.

Table 12.8
Methods for querying edit controls

To perform this action	Call this method
Find out if text has changed	*IsModified*
Retrieve all text	*GetText*
Retrieve a line	*GetLine*
Get the number of lines	*GetNumLines*
Get the length of a given line	*GetLineLength*
Get index of selected text	*GetSelection*
Get a range of characters	*GetSubText*
Count characters before a line	*LineIndex*
Find the line containing an index	*GetLineFromPos*

Text that spans lines in a multiline edit control contains two extra characters for each line break: *carriage return* (#13) and *linefeed* (#10). *TEdit* methods retain the text's formatting when they return text from a multiline edit control. When you insert this text back into an edit control, paste it from the clipboard, write it to a file, or print it to a printer, the line breaks appear as they did in the edit control. Be sure to account for the two extra characters taken up by a line break when you use query methods that get a specified number of characters.

The methods in Listing 12.1 show how to retrieve text, a line of text, and the selected text from an edit control.

Listing 12.1
Retrieving text from edit controls

```
procedure TTestWindow.ReturnText(RetText: PChar);
var TheText: array[0..20] of Char;
begin
  if EC1^.GetText(@TheText, 20) then
    RetText := @TheText
  else RetText := nil;
end;
procedure TestWindow.ReturnLineText(RetText: PChar; LineNum:
  Integer);
var TheText: array[0..20] of Char;
begin
  RetText := nil;
```

```
  with EC1^ do
  if NumLines >= LineNum then
    if LineLength(LineNum) < 11 then
      if GetLine(@TheText, 20, LineNum) then
        RetText := @TheText;
end;

procedure TTestWindow.ReturnSelectedText(RetText: PChar);
var
  TheText: array[0..20] of Char;
  SelStart, SelEnd: Integer;
begin
  with EC1^ do
  begin
    GetSelection(SelStart, SelEnd);
    GetSubText(TheText, SelStart, SelEnd);
  end;
  RetText := TheText;
end;
```

Modifying edit controls

In a traditional data entry program, you might have no need for your program to directly modify an edit control. The user modifies the text and the program reads the text with a query method. However, many other uses of edit controls require that your application explicitly substitute, insert, clear, or select text. ObjectWindows supports these behaviors, plus the ability to force the edit control to scroll.

Table 12.9
Methods for modifying edit controls

To perform this action	Call this method
Delete all text	*Clear*
Delete selected text	*DeleteSelection*
Delete a range of characters	*DeleteSubText*
Delete a line of text	*DeleteLine*
Insert text	*Insert*
Insert text from clipboard	*Paste*
Replace all text	*SetText*
Select a range of text	*SelectRange*
Scroll text	*Scroll*

Example program: EditTest

EditTest is a program that puts up a main window that serves as the parent window for two edit controls, two static controls, and a button. This window is depicted in Figure 12.7.

The complete file EDITTEST.PAS, and its resource file EDITTEST.RES, are on your distribution disks.

When the user clicks the button, *EditTest* copies the selected text from the left edit control (*EC1*) to the right (*EC2*). *EC2* converts the text to uppercase because it has the *es_UpperCase* style. If *EC1* has no selection, *EditTest* copies all the text to *EC2*.

The edit menu supports editing functions in any edit control that currently has the input focus.

Using combo boxes

A combo box control is a combination of two other controls: a list box and an edit control. It serves the same purpose as a list box—it lets the user choose one text item from a scrollable list of text items by clicking the item with the mouse. The edit control, grafted to the top of the list box, provides another selection mechanism, allowing users to type the text of the desired item. If the list box area of the combo box is displayed, the desired item is automatically selected. Type *TComboBox* descends from type *TListBox* and inherits its methods for modifying, querying, and selecting list items. In addition, *TComboBox* provides methods for manipulating the list part of the combo box, which, in some cases, can *drop down* on request.

Three varieties of combo boxes

There are three types of combo boxes: simple, drop down, and drop down list. All combo boxes display their edit area at all times, but some can show and hide their list box areas. Figure 12.8 shows the appearance of the three combo box types as well as that of a list box. Table 12.10 summarizes the properties of each type of combo box.

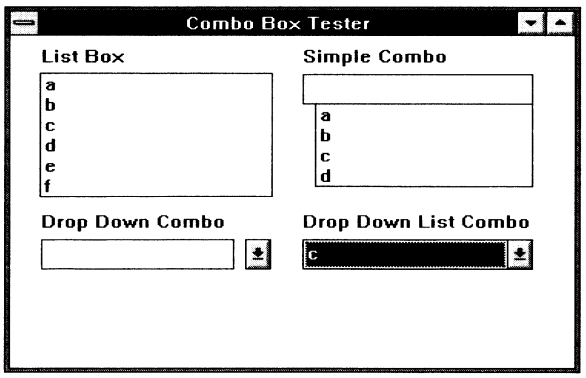


Figure 12.8
The three types of combo boxes and a list box

Table 12.10
Summary of combo box styles

Style	Can hide list	Text must match list
Simple	no	no
Drop down	yes	no
Drop down list	yes	yes

From a user's perspective, these are the distinctions between the different styles of combo boxes:

- Simple combo boxes

  A simple combo box cannot hide its list box area. Its edit area behaves just like an edit control. The user can enter and edit text and the text need not match one of the items in the list. If it does match, the corresponding list item is selected.

- Drop down combo boxes

  A drop down combo box behaves like a simple combo box with one exception. In its initial state, its list area is not displayed. It appears when the user clicks on the icon to the right of the edit area. When they are not being used, they take up much less space than a simple combo box or a list box.

- Drop down list combo boxes

  The list area of a drop down *list* combo box behaves like the list area of a drop down combo box—it appears only when needed. The two combo box types differ in the behavior of their edit areas. Whereas drop down edit areas behave like regular edit controls, drop down list edit areas are limited to displaying only the text from one of its list items. When the edit text matches the item text, no more characters can be entered.

Choosing combo box types

Drop down list combo boxes are useful in cases where no other selection is acceptable besides those listed in the list area. For example, when choosing a printer, you can only choose a printer accessible from your system.

On the other hand, drop down combo boxes can accept entries other than those found in the list. One use of drop down combo boxes is selecting disk files for opening or saving. The user can either search through directories to find the appropriate file in the list, or type the full path name and file name in the edit area, regardless of whether the file name appears in the list area.

Constructing combo boxes

In addition to the six usual control object parameters, the *Init* constructor for *TComboBox* takes a style and a maximum text length as arguments. The *TComboBox* constructor is declared as follows:

```
constructor TComboBox.Init(AParent: PWindowsObject; AnID: Integer;
  X, Y, W, H: Integer; AStyle, ATextLen: Word);
```

All combo boxes constructed with *Init* have the styles *ws_Child*, *ws_Visible*, *cbs_AutoHScroll*, *cbs_Sort* (sorted list), and *ws_VScroll* (vertical scroll bar). The style parameter is one of the standard Windows combo box styles *cbs_Simple*, *cbs_DropDown*, or *cbs_DropDownList*. The text length parameter acts like the corresponding parameter for an edit control, limiting the number of characters users can type in the edit area.

The following lines show a typical combo box constructor, constructing a drop down list combo box with an unsorted list:

```
CB3 := New(PComboBox, Init(@Self, id_CB3, 190, 160, 150, 100,
  cbs_DropDownList, 40));
CB3^.Attr.Style := CB3^.Attr.Style and (not cbs_Sort);
```

Modifying combo boxes

TComboBox defines two methods for showing and hiding the list area of drop down and drop down list combo boxes: *ShowList* and *HideList*. Both procedures take no arguments. You don't need to call these methods to show or hide the list when the user clicks the icon to the right of the edit area. Combo boxes handle that automatically. You only call *ShowList* or *HideList* to force the showing and hiding of the list.

Example program: CBoxTest

The complete file CBOXTEST.PAS is on your distribution disks.

The program *CBoxTest* produces the application shown in Figure 12.8. It uses all three types of combo boxes. *CB1* is a simple combo box, *CB2* is a drop down combo box, and *CB3* is a drop down list combo box. Clicking the Show and Hide buttons forces the showing and hiding of the bottom right combo box, *CB3*, by calling the methods *ShowList* and *HideList*.

Setting control values

To manage complex dialog boxes or windows with many child-window controls, you might typically create a descendant object type to store and retrieve the state of its controls. The state of a control includes the text of an edit control, the position of a scroll bar, and whether a radio button is checked.

Why use transfer buffers?

As an alternative, you can avoid defining a descendant object by defining and supplying a record to represent the state of a window's or a dialog box's controls. This record is called a transfer buffer because it is easy to transfer state information between the buffer and the set of controls.

For example, your program can bring up a modal dialog box and, after closing it, extract information from the transfer buffer about the state of each control. Then, if the user brings up the dialog box again, it can set the controls to their states when the dialog box last closed. In addition, you can set the initial state of each control based on the transfer buffer data. You can also explicitly transfer data in either direction at any time, such as to reset the states of controls to their previous values. A window or modeless dialog box with controls can also use the transfer mechanism to set or retrieve state information at any time.

Associating control objects with control elements is described in Chapter 11, "Dialog box objects."

The transfer mechanism requires the use of ObjectWindows objects to represent the controls for which you would like to transfer data. This means you have to use *InitResource* to associate objects with controls in dialog boxes or dialog windows.

To use the transfer mechanism, you have to do three things:

- Define the transfer buffer
- Define the corresponding window
- Transfer the data

Defining a transfer buffer

The transfer buffer is a record with one field for each control participating in the transfer. A window or dialog can also have controls whose values are not set by the transfer mechanism. For instance, push buttons, which have no states, do not participate in transfers, and neither do group boxes.

The type of control determines the type of field defined for the transfer buffer.

To define a transfer buffer, define a field for each participating control in the dialog or window. It is not necessary to define transfer fields for every control in a dialog or window, only those fields you want to transfer values to and from. This transfer buffer stores one of each type of control, except a push button and a group box:

Listing 12.2
A sample transfer buffer record

```
type
  TSampleTransferRecord = record
    Stat1: array[0..TextLen-1] of Char;              { static text }
    Edit1: array[0..TextLen-1] of Char;          { edit control text }
    List1Strings: PStrCollection;                 { list box strings }
    List1Selection: Integer;              { index of selected string }
    ComboStrings: PStrCollection;                { combo box strings }
    ComboSelection: array[0..TextLen-1] of Char;   { selected string }
    Check1: Word;                                  { check box state }
    Radio1: Word;                               { radio button state }
    Scroll1: TScrollBarTransferRec;         { scroll bar range, etc. }
  end;
```

Each type of control has different information to store. Table 12.11 explains the transfer buffer for each of the standard controls.

Table 12.11
Transfer buffer fields for each control type

Control type	Transfer buffer
Static	A character array up to the maximum length of text allowed, plus the terminating null
Edit	The edit control text buffer, up to the length defined in the *TextLen* field
List box	
single-selection	The collection of strings in the list, plus an integer index to the selected string
multiple-selection	The collection of strings in the list, plus a record containing indexes to all the selected items
Combo box	The collection of strings in the list, plus the selected string
Check box	*Word* values, with values *bf_Unchecked*, *bf_Checked*, and *bf_Grayed* indicating those states
Radio button	Same as check box
Scroll bar	A record of type *TScrollBarTransferRec* stores the range and position of the scroll bar

TScrollBarTransferRec:

```
TScrollBarTransferRec = record
  LowValue: Integer;
  HighValue: Integer;
  Position: Integer;
end;
```

Defining the window

A window or dialog box that uses the transfer mechanism must construct its participating control objects in the exact order in which the corresponding transfer buffer fields are defined. To enable the transfer mechanism for a window or dialog box object, simply set its *TransferBuffer* field to point to a transfer buffer you define.

Using Transfer with a dialog box

Because dialog boxes and dialog windows get their definitions and the definitions of their controls from resources, use the *InitResource* constructor to construct the control objects. For example:

This example uses the TSampleTransferRecord defined in Listing 12.2.

```
type
  TSampleTransferRecord = record
    :
  PParentWindow = ^TParentWindow;
  TParentWindow = object(TWindow)
    Dialog: PDialog;
    XBuffer: TSampleTransferRecord;
      :

constructor TParentWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
var
  Stat1: PStatic;
  Edit1: PEdit;
  List1: PListBox;
  Combo1: PComboBox;
  Check1: PCheckBox;
  Radio1: PRadioButton;
  Scroll1: PScrollBar;
begin
  inherited Init(AParent, ATitle);
  Dialog^.Init(@Self, 'SAMPLE');       { construct dialog box object }
  New(Stat1, InitResource(TheDialog, id_Stat1));   { create controls }
  New(Edit1, InitResource(TheDialog, id_Edit1));
  New(List1, InitResource(TheDialog, id_List1));
  New(Combo1, InitResource(TheDialog, id_Combo1));
  New(Check1, InitResource(TheDialog, id_Check1));
  New(Radio1, InitResource(TheDialog, id_Radio1));
  New(Scroll1, InitResource(TheDialog, id_Scroll1));
  Dialog^.TransferBuffer := @XBuffer;     { point to transfer buffer }
end;
```

Controls constructed with *InitResource* automatically have their transfer mechanisms enabled.

Using Transfer with a window

In the case of a window with controls, use *Init*, rather than *InitResource*, to construct the control objects in the proper order. Control objects constructed with *Init* have their transfer mechanisms initially disabled by default. To enable the mechanism, call *EnableTransfer*:

```
constructor TSampleWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
begin
  inherited Init(AParent, ATitle);
  Edit1 := New(PEdit, Init(@Self, id_Edit1, '', 10, 10, 100, 30,
    40, False));
  Edit1^.EnableTransfer;
end;
```

To explicitly exclude a control from the transfer mechanism, call its *DisableTransfer* method after constructing it.

Transferring the data

In most cases, transferring data to or from a window is fairly automatic, but you can also explicitly transfer data at any time.

Transferring data to a window

Transfer to a window happens automatically when you construct a window or dialog box object. The constructor calls *SetupWindow* to create a screen element to represent the window object and then calls *TransferData* to load any data from the transfer buffer. The window's *SetupWindow* iteratively calls *SetupWindow* for each of its child windows as well, so each of the child windows has a chance to transfer its data. Because the parent window sets up its child windows in the order it constructed them, the data in the transfer buffer must appear in that same order.

Control objects' *SetupWindow* methods call *TransferData* if the control object was constructed with *InitResource* or if the parent window explicitly enabled the transfer mechanism by calling *EnableTransfer* for the control.

Transferring data from a dialog box

When a modal dialog box receives a command message with a control ID of *id_OK*, it automatically transfers data from the controls into the transfer buffer. Usually this message indicates that the user clicked an OK button to end the dialog, so the dialog automatically updates its transfer buffer. Then, if you execute the dialog again, the dialog box transfers the current data to the controls.

Transferring data from a window

You can explicitly transfer data in either direction at any time. For example, you might want to transfer data out of controls in a window or modeless dialog. Or you might want to reset the state of the controls using the data in the transfer buffer in response to the user clicking a Reset button.

Use the *TransferData* method in either case, passing the *tf_SetData* constant to transfer from the buffer to the controls and the *tf_GetData* constant to transfer in the other direction. For example, you might want to call *TransferData* in the *Destroy* method of a window object:

```
procedure TSampleWindow.Destroy;
begin
  TransferData(tf_GetData);
  inherited Destroy;
end;
```

Supporting transfer for custom controls

You might want to modify the way a particular control transfers its data or include a new control you define in the transfer mechanism. In either case, you simply need to write a *Transfer* method for your control object which, if the flag parameter is *tf_GetData*, copies data from the control to the location specified by the passed pointer. If the flag parameter is *tf_SetData*, copy the data at the passed pointer into the control. As an example, here is *TStatic.Transfer*:

```
function TStatic.Transfer(DataPtr: Pointer; TransferFlag: Word):
Word;
begin
  if TransferFlag = tf_GetData then
    GetText(DataPtr, TextLen)
  else if TransferFlag = tf_SetData then
    SetText(DataPtr);
  Transfer := TextLen;
end;
```

The *Transfer* method should always return the number of bytes transferred.

Example program: TranTest

The complete file TRANTEST.PAS is on your distribution disks.

The *TranTest* program's main window produces a modal dialog with fields for the user to enter name and address information. It uses a transfer buffer to store this information and display it in the dialog's controls when the dialog is again executed. Notice that we did not need to define a new dialog object type to set and retrieve the dialog's data. Notice also that we directly manipulate data in the transfer buffer so that the static control in the first appearance of the dialog reads, "First Mailing Label," while in each additional appearance, it reads, "Subsequent Mailing Label."

Using custom controls

Windows provides mechanisms that enable you to create your own kinds of controls, and ObjectWindows makes it easy to create objects that take advantage of custom controls. This section discusses how you can use the Borland Windows Custom Controls that give Borland's Windows applications their distinctive look, then describes how to create your own unique controls.

Borland Windows Custom Controls

Borland Windows Custom Controls (BWCC) provide the distinctive look of Borland's Windows applications. The key features of BWCC are

- bitmapped buttons
- gray "chiseled steel" backgrounds for dialog boxes
- 3-D check boxes and radio buttons

ObjectWindows gives you easy access to BWCC, so you can incorporate the Borland look in your own applications.

Using standard BWCC

ObjectWindows makes it simple to add BWCC to your Windows applications. Just add the BWCC unit to your main program's **uses** clause:

```
uses BWCC;
```

Using the BWCC unit automatically enables you to do two things:

- Use BWCC controls loaded from resources
- Create BWCC controls in your program

For example, using Resource Workshop you can create dialog box resources that use the Borland custom controls. Including BWCC in your **uses** clause ensures that your program knows where to find the dynamic-link library (BWCC.DLL) that contains the code that makes BWCC work.

In addition, once you've included the BWCC unit, any control objects you create within your own program will have the BWCC look and feel.

BWCC features

BWCC adds some new styles to the standard Windows control styles, but it also has some conventions and expectations. If you create all your controls from resources, you won't need to worry about these. When you construct control objects in your code, however, you might need to use some of the new styles and follow the conventions.

Extending BWCC

BWCC provides bitmap buttons for all the standard Windows buttons. That is, there are bitmaps provided for each of the buttons for which Windows provides a standard ID: *id_Abort*, *id_Cancel*, *id_Ignore*, *id_No*, *id_Ok*, *id_Retry*, and *id_Yes*.

Creating bitmap buttons

You can provide your own bitmaps for push buttons in your applications. All you have to provide are six bitmap resources numbered relative to your push button control's ID. For example, if you want to create a bitmap push button with the ID *id_MyButton*, you'll create bitmap resources with resource IDs 1000 + *id_MyButton*, 2000 + *id_MyButton*, 3000 + *id_MyButton*, 4000 + *id_MyButton*, 5000 + *id_MyButton*, and 6000 + *id_MyButton*. Table 12.12 shows what each bitmap represents.

Table 12.12
Bitmap resources for BWCC push buttons

Image	VGA resource ID	EGA resource ID
Button focused	1000 + ID	2000 + ID
Button pressed	3000 + ID	4000 + ID
Button unfocused	5000 + ID	6000 + ID

VGA push button bitmaps should be 63 pixels wide and 39 pixels high. EGA versions should be 63 pixels wide and 29 pixels high.

Text should be 8-point Helvetica, and the focused image should have a dotted box around the text. Figure 12.9 shows a set of bitmaps for a button with an ID of 201.

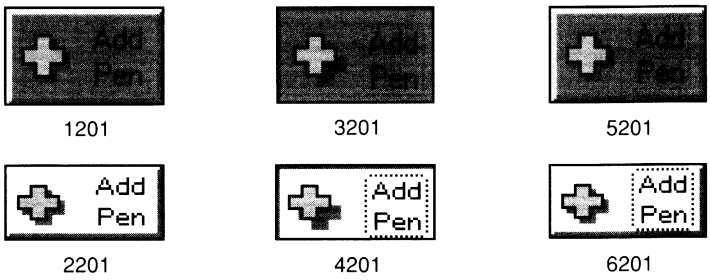


Figure 12.9
Bitmap resources for a BWCC push button, ID 201

Creating your own custom controls

The easiest way to create a custom control is actually to create a window that acts like a control but isn't a control at all. This is the approach taken in the *Steps* program in Part 1 of this manual. The same approach *Steps* uses for its pen palette object could be used to create a tool bar object, for example. Such "controls" are descendants of *TWindow*, rather than from *TControl*, because *TControl* handles only the standard Windows controls.

Window classes are explained in Chapter 10, "Window objects."

The other standard way to make a custom control is to create a new window class in a dynamic-link library. You can then create ObjectWindows objects that make use of the new class. Resource Workshop can also make use of custom controls created in DLLs. See the *Resource Workshop User's Guide* for information on using custom controls in dialog box resources.

C H A P T E R

13

Data validation

ObjectWindows gives you several flexible ways to validate the information a user types into an edit control by associating validator objects with the edit control objects. Using validator objects makes it easy to add validation to existing ObjectWindows applications or to change the way a field validates its data.

This chapter covers three topics related to data validation:

- The three kinds of data validation
- Using data validator objects
- How validators work

Data validation is handled by the *CanClose* method of interface objects. You can validate the contents of any particular edit control or data screen at any time by calling the object's *CanClose* method, but ObjectWindows also provides mechanisms to automate data validation. Most of the time, you'll find that data validation takes almost no effort on your part as programmer.

The three kinds of data validation

There are three distinct types of data validation, and Object-Windows supports all three directly. The three kinds of data validation are

- Filtering input
- Validating each item
- Validating complete screens

Note that these methods are not mutually exclusive. A number of the standard validators combine the different techniques in a single validator.

It's important to remember that the validation is handled by the validator object, *not* by the edit control object. If you've already created a customized edit control object for a specialized purpose, you've probably already duplicated capability that's already built into edit controls and their validators.

The "How validators work" section of this chapter describes the various ways in which edit control objects automatically call on validator objects.

Filtering input

The simplest way to ensure that a field contains valid data is to ensure that the user can only *type* valid data. ObjectWindows provides *filter* validators that enable you to restrict the characters the user can type. For example, a numeric input field might restrict the user to typing only numeric digits.

ObjectWindows' filter validator object provides a generic mechanism for limiting which characters a user can type in a given edit control. Picture validator objects can also control the formatting and types of characters a user can type.

Validating each field

Sometimes you'll find it necessary to ensure that the user types valid input in a particular field before moving to the next field. This approach is often called "validate on Tab," since pressing *Tab* is the usual way to move the input focus in a data entry screen.

An example would be an application that performs a lookup from a database, where the user types in some kind of key information in a field, and the application responds by retrieving the appropriate record and filling the rest of the fields. In such a case, your application needs to check that the user has typed the proper information in the key field before acting on that key.

Validating full screens

You can handle validation of full data screens in three different ways:

- Validating modal windows
- Validating on focus change
- Validating on demand

Validating modal windows

When a user closes a modal window, the window automatically validates all its subviews before closing, unless the closing command was *cmCancel*. To validate its subviews, the window calls each subview's *CanClose* method, and if each returns *True*, the window can close. If any of the subviews returns *False*, the window won't be allowed to close.

A modal window with invalid data can only be canceled until the user provides valid data.

Validating on demand

You can tell a window to validate all its subviews at any time by calling its *CanClose* method. Calling *CanClose* essentially asks the window "If I told you to close right now, would all your fields be valid?" The window calls the *CanClose* methods of all its child windows in the order of insertion and returns *True* if all of them return *True*.

Calling *CanClose* does not obligate you to actually close the window. For example, you might call *CanClose* when the user presses a Save button, ensuring the validity of the data before saving it.

You can validate any window, modal or modeless, at any time. Only modal windows have automatic validation on closing, however. If you use modeless data entry windows, you'll need to ensure that your application calls the window's *CanClose* method before acting on entered data.

Using a data validator

Using a data validator object with an edit control takes only two simple steps:

- Constructing the validator object
- Assigning the validator to an edit control

Once you've constructed the validator and associated it with an edit control, you never need to interact with the validator object directly. The edit control knows when to call validator methods at the appropriate times.

Constructing validator objects

Since validators are not interface objects, their constructors require only enough information to establish the validation criteria. For example, a numeric range validator object takes two parameters: the minimum and maximum values in the valid range, as show here.

```
constructor TRangeValidator.Init(AMin, AMax: Integer);
```

Adding validation to edit controls

Every edit control object has a field called *Validator*, set to **nil** by default, that can point to a validator object. If you don't assign an object to *Validator*, the edit control behaves as described in Chapter 12, "Control objects." Once you assign a validator by calling *SetValidator*, the edit control automatically checks with the validator when processing key events and when called on to validate itself.

Normally you'll construct and assign the validator in a single statement, as shown in Listing 13.1.

Listing 13.1
Adding data validation to an edit control

```
  ⋮
                                        { make 3-char edit control }
Ed := New(PEdit, Init(@Self, id_Me, '', 10, 10, 50, 30, 3, False));
Ed^.SetValidator(New(PRangeValidator, Init(100, 999)));
  ⋮
```

How validators work

ObjectWindows supplies several kinds of validator objects that should cover most of your data validation needs. You can also derive your own validators from the abstract validator types.

This section covers the following topics:

- The virtual methods of a validator
- The standard validator types

The methods of a validator

Every validator object inherits four important methods from the abstract validator object type *TValidator*. By overriding these methods in different ways, the various descendant validators perform their specific validation tasks. If you're going to modify the standard validators or write your own validation objects, you need to understand what each of these methods does and how edit controls use them.

The four validation methods are

- *Valid*
- *IsValid*
- *IsValidInput*
- *Error*

The only methods called from outside the object are *Valid* and *IsValidInput*. *Error* and *IsValid* are only called by other validator methods.

Checking for valid data

The main external interface to data validator objects is the method *Valid*. Much like the *CanClose* methods of interface objects, *Valid* is a Boolean function that returns *True* only if the string passed to it is valid data. One component of an edit control's *CanClose* method is calling the validator's *Valid* method, passing the edit control's current text.

When using validators with edit controls, you should never need to either call or override the validator's *Valid* method. By default, *Valid* returns *True* if the method *IsValid* returns *True*; otherwise, it calls *Error* to notify the user of the error and returns *False*.

Validating a complete line

Validator objects have a virtual method called *IsValid* that takes a string as its only parameter and returns *True* if the string represents valid data. *IsValid* is the method that does the actual validation, so if you create your own validator objects, you'll almost definitely override *IsValid*.

Note that you don't call *IsValid* directly. Use *Valid* to call *IsValid*, because *Valid* calls *Error* to alert the user if *IsValid* returns *False*. Be

sure to keep the validation role separate from the error reporting role.

Validating keystrokes

When an edit control object recognizes a keystroke event meant for it, it calls its validator's *IsValidInput* method to ensure that the typed character is a valid entry. By default, *IsValidInput* methods always return *True*, meaning that all keystrokes are acceptable, but some descendant validators override *IsValidInput* to filter out unwanted keystrokes.

For example, range validators, which are used for numeric input, return *True* from *IsValidInput* only for numeric digits and the characters '+' and '-'.

IsValidInput takes two parameters. The first parameter is a **var** parameter, holding the current input text. The second parameter is a Boolean value indicating whether the validator should apply filling or padding to the input string before attempting to validate it. *TPictureValidator* is the only one of the standard validator objects that makes use of the second parameter.

Reporting invalid data

The virtual method *Error* alerts the user that the contents of the edit control don't pass the validation check. The standard validator objects generally present a simple message box notifying the user that the contents of the input are invalid and describing what proper input would be.

For example, the *Error* method for a range validator object creates a message box indicating that the value in the edit control is not between the indicated minimum and maximum values.

Although most descendant validator objects override *Error*, you should never call it directly. *Valid* calls *Error* for you if *IsValid* returns *False*, which is the only time *Error* needs to be called.

The standard validators

ObjectWindows includes six standard validator object types, inlcuding an abstract validator and the following five specific validator types:

- Filter validator
- Range Validator
- Lookup validator

- String lookup validator
- Picture validator

The abstract validator

The abstract type *TValidator* serves as the base type for all the validator objects, but it does nothing useful by itself. You never create an instance of *TValidator*. Essentially, *TValidator* is a validator to which all input is always valid: *IsValid* and *IsValidInput* always return *True*, and *Error* does nothing. Descendant types override *IsValid* and/or *IsValidInput* to define which values actually are valid.

You can use *TValidator* as a starting point for your own validator objects if none of the other validation types are appropriate starting points.

Filter validators

Filter validators are a simple implementation of validators that only check input as the user types it. The filter validator constructor takes one parameter, a set of valid characters:

```
constructor TFilterValidator.Init(AValidChars: TCharSet);
```

TFilterValidator overrides *IsValidInput* to return *True* only if all characters in the current input string are contained in the set of characters passed to the constructor. The edit control only inserts characters if *IsValidInput* returns *True*, so there is no need to override *IsValid*: Because the characters made it through the input filter, the complete string is valid by definition.

Descendants of *TFilterValidator* such as *TRangeValidator* can combine filtering of input with other checks on the completed string.

Range validators

The range validator *TRangeValidator* is a straightforward descendant of *TFilterValidator* that accepts only numbers and adds range checking on the final results. The constructor takes two parameters that define the minimum and maximum valid values:

```
constructor TRangeValidator.Init(AMin, AMax: Integer);
```

The range validator constructs itself as a numeric filter validator, accepting only the digits '0'..'9' and the plus and minus characters. The inherited *IsValidInput*, therefore, ensures that only numbers filter through. *TRangeValidator* then overrides *IsValid* to return *True* only if the entered numbers are a valid integer within the range defined in the constructor. The *Error* method displays a message box indicating that the entered value is out of range.

Lookup validators

The abstract lookup validator *TLookupValidator* provides the basis for a common type of data validator, one which compares the entered value with a list of acceptable items in order to determine validity.

For an example of a working lookup validator, see the section on string lookup validation.

TLookupValidator is an abstract type that you never use as it stands, but it makes one important change and one addition to the standard abstract validator.

The one new method introduced by *TLookupValidator* is called *Lookup*. By default, *Lookup* just returns *False*, but when you derive a descendant lookup validator type, you override *Lookup* to compare the passed string with a list, returning *True* if the string contains a valid entry.

TLookupValidator overrides *IsValid* to return *True* only if *Lookup* returns *True*. In descendant lookup validator types, you should *not* override *IsValid*, but rather override *Lookup*.

String lookup validators

TStringLookupValidator is a working example of a lookup validator, comparing the string passed from the edit control with the items in a string list. If the passed string occurs in the list, the string lookup validator's valid method returns *True*. The constructor takes only one parameter, the list of valid strings:

```
constructor TStringLookupValidator.Init(AStrings: PStringCollection);
```

To use a different string list after constructing the string lookup validator, pass the new list to the validator's *NewStringList* method, which disposes of the old list and installs the new list.

TStringLookupValidator overrides *Lookup* and *Error*, so that *Lookup* returns *True* if the passed string is in the string collection and *Error* displays a message box indicating that the string wasn't in the list.

Picture validators

Picture validators compare the string typed by the user with a *picture* or template that describes the format of valid input. The pictures used are compatible with those used by Borland's Paradox relational database to control user input. Constructing a picture validator takes two parameters: a string holding the template image and a Boolean value indicating whether to automatically fill in literal characters in the picture:

The syntax for pictures is described in the reference for TPXPictureValidator.Picture.

```
constructor TPictureValidator.Init(const APic: String;
  AAutoFill: Boolean);
```

TPictureValidator overrides *Error*, *IsValidInput*, and *IsValid*, and adds a new method, *Picture*. The changes to *Error* and *IsValid* are simple: *Error* displays a message box indicating what format the string should have, and *IsValid* returns *True* only if the function *Picture* returns *True*, allowing you to derive new kinds of picture validators by overriding only the *Picture* method. *IsValidInput* checks characters as the user types them, allowing only those allowed by the picture format, and optionally filling in literal characters from the picture.

The *Picture* method tries to format the given input string according to the picture format and returns a value indicating the degree of its success: complete, incomplete, or error.

C H A P T E R

14

MDI objects

The multiple document interface is an interface standard for Windows applications that allows the user to simultaneously work with many open documents. A document, in this sense, is usually a file-specific task, such as editing a text file or working on a spreadsheet file. The MDI standard is also part of IBM's Common User Access specification.

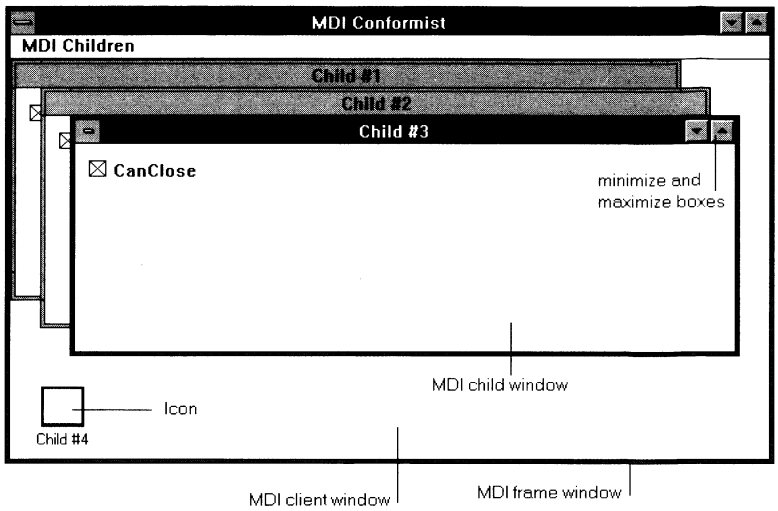
ObjectWindows provides objects that ease writing MDI applications.

What is an MDI application?

The invisible client window is important because it provides behind-the-scenes management of the MDI child windows.

Certain components are present in every MDI application. Most evident is the main window, called the *frame window*. Within the frame window's client area is an invisible window, the MDI client window, which holds child windows called *MDI child windows*. Figure 14.1 shows an MDI application and its parts.

Figure 14.1
The components of an MDI application



The child window menu

The frame window's menu bar has a menu, often labeled Window, which controls the MDI child windows. This *child-window menu* usually has items such as Tile, Cascade, Arrange, and Close All. The name of each open MDI child window is automatically appended to the end of this menu, with the currently selected window checked.

MDI child windows

Each MDI child window has some characteristics of an overlapped window. It can be maximized to the full size of its MDI client window or minimized to an icon, which sits above the bottom edge of the frame window. MDI child windows never appear outside the borders of their frame window. MDI child windows cannot have menus, so all functions are controlled by the frame window's menu. The caption of each MDI child window is often the name of the open file associated with that window, although this behavior is optional and under program control. You can think of an MDI application as a mini-Windows session, complete with many applications represented by windows or icons.

MDI windows in ObjectWindows

ObjectWindows defines the objects *TMDIWindow* and *TMDIClient* to represent MDI frame and client windows. Both objects descend from *TWindow*, but *TMDIClient* is actually a control and has much in common with *TControl*. In an ObjectWindows MDI application, the frame window owns its MDI client window and stores it in its *ClientWnd* field. The frame window also holds each of its MDI child windows in its *ChildList* linked list. The MDI child windows can be instances of any object type descending from *TWindow*.

TMDIWindow's methods deal mainly with construction and management of MDI child windows and the MDI client window and with processing menu selections. *TMDIClient*'s primary role is behind-the-scenes management of MDI child windows. In designing MDI applications, you generally derive new types from *TMDIWindow* and *TWindow* for your frame and child windows, respectively.

Building an MDI application

Building an MDI application is relatively easy in ObjectWindows:

- Construct an MDI main window
- Set up a child window menu
- Teach the main window to create MDI children

The MDI window handles all the MDI-specific functions for you, and your application-specific functions can go into the child windows.

Constructing an MDI frame

The MDI frame window is always an application's main window, so it is constructed in the *InitMainWindow* method of the application object. There are two aspects of an MDI frame that make it different from other main windows, however:

- An MDI frame is *always* a main window, so it never has a parent. Therefore, *TMDIWindow.Init* does not take a parent window pointer parameter.

- An MDI frame window *must* have a menu, so *Init*'s second parameter is a handle to a menu. With non-MDI main windows, derived from *TWindow*, you define *Init* to set *Attr.Menu* to a valid menu handle. *TMDIWindow.Init* sets *Attr.Menu* for you.

A typical *InitMainWindow* for an MDI application would look like this:

```
procedure TMDIApplication.InitMainWindow;
begin
  MainWindow := New(PMyFrame, Init('Frame Title',
    LoadMenu(HInstance, 'MenuName'));
end;
```

Assuming *TMyFrame* is a descendant of *TMDIWindow*, this would create an MDI frame window with the title 'Frame Title' and a menu bar defined by a resource called 'MenuName'.

Creating a child-window menu

The frame window's menu must include an MDI-style child-window menu. Opening an MDI child window appends its caption to the child-window menu, and closing the child window removes it from the list. This allows a user to activate any child window, even if it is not visible.

The frame window needs to know which top-level menu item is its child-window menu. *TMDIWindow* stores an integer value in the *ChildMenuPos* field, indicating which item on the menu bar gets the child list. *TMDIWindow.Init* initially sets *ChildMenuPos* to zero, indicating the leftmost top-level menu item. However, you can redefine *Init* for your *TMDIWindow*-descended types to reset *ChildMenuPos*:

```
constructor TMyMDIWindow.Init(ATitle: PChar; AMenu: HMenu);
begin
  inherited Init(ATitle, AMenu);
  ChildMenuPos := 1;
end;
```

TMDIWindow.Init also calls *InitClientWindow* to construct the *TMDIClient* object to serve as its MDI client window. *TMDIWindow.SetupWindow* creates the MDI client window.

Constructing an MDI child window

TMDIWindow defines an automatic response method called *CreateChild* that is invoked upon a menu selection bearing the command ID *cm_CreateChild*. Usually this menu item is called New or Create. As defined by *TMDIWindow*, *CreateChild* constructs and creates an MDI child window of type *TWindow* by calling *TMDIWindow.InitChild*. To specify the correct child window type (any descendant of *TWindow*), redefine *InitChild* for your MDI frame window type:

```
function TMyMDIWindow.InitChild: PWindowsObject;
begin
  InitChild := New(PMyChild, Init(@Self, 'New Child Window'));
end;
```

Automatic child windows

You might want your frame window to produce *one* MDI child window when it first appears. For this child window, you must explicitly specify its size. Unlike other child windows, MDI child windows must be constructed and created from within the MDI frame window's *SetupWindow* method, rather than from within *Init*. You also have to explicitly create the child window's screen element by calling *MakeWindow*:

```
procedure TMyMDIWindow.SetupWindow;
var
  ARect: TRect;
  NewChild: PMyChild;
begin
  inherited SetupWindow;
  NewChild := PMyChild(InitChild);
  GetClientRect(HWindow, ARect);
  with NewChild^.Attr, ARect do
  begin
    W := (right * 4) div 5;
    H := (bottom * 3) div 5;
    Title := 'Child #1';
  end;
  Application^.MakeWindow(NewChild);
end;
```

In some applications, you want to create MDI child windows in response to more than one menu choice. For example, New and Open menu choices in a file editor might both bring up a new child window with the file name as caption. In that case, define

automatic response methods to construct the child window. ObjectWindows defines the commands *cm_MDIFileOpen* and *cm_MDIFileNew* so you can differentiate from the standard *cm_FileOpen* and *cm_FileNew*.

Manipulating child windows

The ObjectWindows MDI window types provide methods for manipulating the MDI child windows of an MDI application. While much of the underlying work is done by *TMDIClient*, all of the functionality and data is accessible through *TMDIWindow* methods.

TMDIWindow has methods that automatically respond to the standard MDI menu selections: Tile, Cascade, Arrange Icons, and Close All. These methods expect command-based messages with predefined menu ID constants. Be sure to use these IDs when building a child-window menu resource:

Table 14.1
Standard MDI actions, commands, and methods

Action	Menu ID constant	TMDIWindow method
Tile	*cm_TileChildren*	*CMTileChildren*
Cascade	*cm_CascadeChildren*	*CMCascadeChildren*
Arrange Icons	*cm_ArrangeChildIcons*	*CMArrangeChildIcons*
Close All	*cm_CloseChildren*	*CMCloseChildren*

TMDIWindow's response methods, such as *CMTileChildren*, call other *TMDIWindow* methods, such as *TileChildren*. These methods call *TMDIClient* methods of the same name, such as *TMDICLient^.TileChildren*. To redefine any of this automatic behavior, override *TMDIWindow.TileChildren* or the other *TMDIWindow* methods. It is not appropriate for MDI child windows to respond to command messages generated by the child-window menu.

Customizing window activation

The user of an MDI application is free to activate any open or minimized MDI child window. However, you might want to take some action when the user deactivates one child window by activating another. For example, the frame window's menu might reflect the current state of the active child window through graying or checking. Every time a child window becomes active or inactive, it receives the Windows message *wm_MDIActivate*. By defining a message-response method for this message for the child window, you can track which child window is active and respond accordingly.

Processing messages in an MDI application

As with regular parent and child windows, Windows command-based and child-ID-based messages first come to the child window for a chance to intercept and process them. Then, the messages go to the parent window. In the case of MDI applications, however, the messages come to the currently active MDI child window, then to the MDI client window, and finally to the MDI frame window (which is the common parent to all of the MDI child windows). This way, the frame window's menu can be used to control activity in the currently active MDI child window. Then the client and frame windows have a chance to respond.

Sample MDI application

The program *MDITest* produces the MDI-compliant application in Figure 14.1. The complete file, MDITEST.PAS, can be found on your distribution disks.

C H A P T E R

15

Printing objects

One of the most difficult aspects of Windows programming is getting output to the printer. ObjectWindows makes printing easier through the use of objects that encapsulate the behavior of the printer device and the printout itself.

This chapter describes the basics of the printing process, then describes the following printing tasks:

- Constructing a printer object
- Creating the printout
  - Printing a document
  - Printing the contents of a window
- Sending the printout to a printer
- Choosing a different printer
- Configuring the printer

Why is printing difficult?

On one hand, printing in Windows is quite simple. You use the same GDI functions to generate printed output that you use to put images on the screen. To write text you use *TextOut*, to draw a rectangle you call *Rectangle*.

On the other hand, the process is complicated because Windows requires the application to "talk" directly with the printer drivers

through calls to *Escape* or by retrieving the address of *DeviceMode* or *ExtDeviceMode*. It is further complicated by Windows requiring the application to retrieve the name of the device driver from the WIN.INI file. In addition, there is more variation in capabilities and resolutions among printers than there is among video devices.

ObjectWindows can't take all the obstacles out of your way, but it makes the process of printing much simpler and easier to understand.

Printing in ObjectWindows

ObjectWindows' *OPrinter* unit provides two objects to simplify printing: *TPrinter* and *TPrintout*. *TPrinter* encapsulates access to the printer drivers. It is capable of configuring the printing by bringing up a dialog that allows the user to select the desired printer as well as setting the current settings for printing, such as the graphics resolution and landscape or portrait orientation.

TPrintout encapsulates the task of printing a document. Its relationship to the printer corresponds, roughly, to *TWindow's* relationship to the screen. Drawing on the screen happens in the *Paint* method of the *TWindow* object, where writing to the printer happens in the *PrintPage* method of the *TPrintout* object. To print something on the printer, the application passes an instance of *TPrintout* to an instance of *TPrinter's Print* method.

Constructing a printer object

In most cases, an application will only need to access one printer at a time. The easiest way to handle this is to give the main window object a field called *Printer* (of type *PPrinter*), which other objects in the program call for their printing needs. To make the printer available, make *Printer* point to an instance of *TPrinter*.

For most applications, this is simple. The application's main window initializes a printer object that uses the default printer specified in WIN.INI:

```
constructor TSomeWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  inherited Init(AParent, ATitle);
   ⋮
```

```
  Printer := New(PPrinter, Init);
end;
```

In some cases, you might have applications that use different printers from different windows simultaneously. In that case, construct a printer object in the constructors of each of the appropriate windows, then change the printer device for one or more of the printers. If the program uses different printers but not at the same time, it's probably best to use the same printer object and select different printers as needed.

Although you might be tempted to override the *TPrinter* constructor to use a printer other than the system default, the recommended procedure is to always use the default constructor, then change the device associated with the object. See "Choosing a different printer" on page 214.

Creating the printout

Windows graphics functions are explained in Chapter 17, "The Graphics Device Interface."

The only tricky part of printing in ObjectWindows is creating the printout object. The process is similar to writing a *Paint* method for a window object: You use Windows' graphics functions to generate the image you want on a device context. The window object's display context handles your interactions with the screen device; the printout object's device context insulates you from the printer device in much the same way.

To create a printout object, derive a new object type from *TPrintout* that overrides *PrintPage*. In very simple cases, that's all you do. If the document is more than one page long, you also override *HasNextPage* to return *True* while there is another page to be printed. The current page number is passed as a parameter to *PrintPage*.

The printout object has fields that hold the size of the page and a device context that is already initialized to render to the printer. The printer object sets those values by calling the printout object's *SetPrintParams* method. Use the printout object's device context in any calls to Windows graphics functions.

The *OPrinter* unit includes two specialized printout objects which show the range of complexity of printouts. *TWindowPrintout*, which prints the contents of a window, is extremely simple. *TEditPrintout*, which prints the contents of an edit control, is very complex because it has many more options.

Printing a document

Windows sees a printout as a series of pages, so the job of your printout object is to turn a document into a series of page images for Windows to print. Just as you use window objects to paint images for Windows to display on the screen, you use printout objects to paint images for Windows to print on the printer.

ObjectWindows provides an abstract printout object, *TPrintout*, from which you derive useful printout objects. You need to redefine only a few methods in *TPrintout*.

Your printout object needs to be able to do three things:

- Set print parameters
- Count its pages
- Draw each page on a device context
- Indicate if there are more pages

The rest of this section refers to the sample program *PrnTest*, included on your distribution disks as PRNTEST.PAS. *PrnTest* reads a text file into a collection of strings, then prints the document on command. *PrnTest*'s printout object is declared as follows:

```
type
  PTextPrint = ^TTextPrint;
  TTextPrint = object(TPrintout)
    TextHeight, LinesPerPage, FirstOnPage, LastOnPage: Integer;
    TheLines: PCollection;
    constructor Init(ATitle: PChar; TheText: PPCharCollection);
    function GetDialogInfo(var Pages: Integer): Boolean; virtual;
    function HasNextPage(Page: Word): Boolean; virtual;
    procedure SetPrintParams(ADC: HDC; ASize: TPoint); virtual;
    procedure PrintPage(Page: Word; var Rect: TRect;
      Flags: Word); virtual;
  end;
```

Setting print parameters

Before asking your document to actually print itself, the printer object gives your document the chance to paginate itself, by calling two of the printout object's methods: *SetPrintParams* and then *GetDialogInfo*.

The *SetPrintParams* function is your printout's time to initialize any data structures it might need to produce efficient printout of

individual pages. *SetPrintParams* is your printout's first chance to access the device context and page size for the printer. Listing 15.1 shows an example of an overridden *SetPrintParams* method.

If you override *SetPrintParams*, be sure to call the inherited method, which sets the printout object's field values.

Counting pages

After calling *SetPrintParams*, the printer object calls the Boolean *GetDialogInfo* method. *GetDialogInfo* sets up information needed to display a dialog box that lets the user select page ranges before printing. There are two aspects to *GetDialogInfo*: counting pages and indicating whether to display the dialog box.

GetDialogInfo takes a single **var** parameter, *Pages*, that it should set to the number of pages in the document, or to zero if it can't calculate the number of pages. The return value is *True* if you want to display the dialog box or *False* to suppress the dialog box.

The default *GetDialogInfo* sets *Pages* to zero and returns *True*, meaning that it doesn't know how many pages it might produce, and that a dialog box will appear before printing. You generally override *GetDialogInfo* to set *Pages* to the number of pages in the document and return *True*.

For example, *PrnTest* calculates how many lines of text in the selected font can fit within the print area in its *SetPrintParams*, then uses that number to calculate how many pages it will need to print in *GetDialogInfo*:

Listing 15.1
Counting printout pages

```
procedure TTextPrint.SetPrintParams(ADC: HDC; ASize: TPoint);
var TextMetrics: TTextMetric;
begin
  inherited SetPrintParams(ADC, ASize);           { Set DC and Size }
  GetTextMetrics(DC, TextMetrics);              { get text size info }
  TextHeight := TextMetrics.tmHeight;         { calculate line height }
  LinesPerPage := Size.Y div TextHeight;        { and lines per page }
end;

function TTextPrint.GetDialogInfo(var Pages: Integer): Boolean;
begin
  Pages := TheLines^.Count div LinesPerPage + 1;
  GetDialogInfo := True;        { display dialog box before printing }
end;
```

Printing each page

Once the printer object has given the document a chance to paginate itself, it proceeds to call the printout object's *PrintPage* method for each page to be printed. The process of printing out just the part of the document that belongs on the given page is similar to deciding which portion gets drawn on a scrolling window. For example, note the similarity between *PrnTest*'s window-painting and page-printing methods:

Listing 15.2
Painting and printing text

```
procedure TTextWindow.Paint(PaintDC: HDC;
  var PaintInfo: TPaintStruct);
var
  Line: Integer;
  TextMetrics: TTextMetric;
  TheText: PChar;

  function TextVisible(ALine: Integer): Boolean;
  begin
    with Scroller^ do
      TextVisible := IsVisibleRect(0, (ALine div YUnit) + YPos,
        1, Attr.W div YUnit);
  end;

begin
  GetTextMetrics(PaintDC, TextMetrics);
  Scroller^.SetUnits(TextMetrics.tmAveCharWidth,
    TextMetrics.tmHeight);
  Line := 0;
  while (Line < FileLines^.Count) and TextVisible(Line) do
  begin
    TheText := PChar(FileLines^.At(Line));
    if TheText <> nil then
      TextOut(PaintDC, 0, Line * Scroller^.YUnit, TheText,
        StrLen(TheText));
    Inc(Line);
  end;
end;

procedure TTextPrint.PrintPage(Page: Word; var Rect: TRect;
  Flags: Word);
var
  Line: Integer;
  TheText: PChar;
begin
  FirstOnPage := (Page - 1) * LinesPerPage;
  LastOnPage := (Page * LinesPerPage) - 1;
  if LastOnPage >= TheLines^.Count then
    LastOnPage := TheLines^.Count - 1;
  for Line := FirstOnPage to LastOnPage do
```

```
  begin
    TheText := TheLines^.At(Line);
    if TheText <> nil then
      TextOut(DC, 0, (Line - FirstOnPage) * TextHeight, TheText,
        StrLen(TheText));
  end;
end;
```

There are two issues to keep in mind when writing *PrintPage* methods:

- Device independence. Make sure your code doesn't make assumptions about scale, aspect ratio, or colors. Those properties can vary between different video and printing devices, so you should remove any device dependencies from your code.
- Device capabilities. Although most video devices support all GDI operations, some printers do not. For example, many print devices, such as plotters, do not accept bitmaps at all. Others support only certain operations. When performing complex output tasks, your code should call the Windows API function *GetDeviceCaps*, which returns important information about the capabilities of a given output device.

Indicating further pages

Printout objects have one last duty: to indicate to the printer object whether there are printable pages beyond a given page. The *HasNextPage* method takes a page number as a parameter and returns a Boolean value indicating whether further pages exist. By default, *HasNextPage* always returns *False*. In order to print multiple pages, your printout object needs to override *HasNextPage* to return *True* if the document has more pages to print and *False* if the parameter passed is the last page.

For example, *PrnTest* compares the number of the last line printed with the number of the last line in the file to determine whether more pages need to be printed:

```
function TTextPrint.HasNextPage(Page: Word): Boolean;
begin
  HasNextPage := LastOnPage < TheLines^.Count - 1;
end;
```

Be sure that *HasNextPage* returns *False* at some point. If *HasNextPage* always returns *True*, printing goes into an endless loop.

Other printout considerations

Printout objects have several other methods you can override as needed. *BeginPrinting* and *EndPrinting* are called before and after any documents are printed, respectively. If you need special setup code, you can put it in *BeginPrinting* and undo it in *EndPrinting*.

Printing of pages takes place sequentially. That is, the printer calls *PrintPage* for each page in sequence. Before the first call to *PrintPage*, however, the printer object calls *BeginDocument*, passing the numbers of the first and last pages it will print. If your document needs to prepare to begin printing at a page other than the first, override *BeginDocument*. The corresponding method, *EndDocument*, is called after the last page prints.

The other method you might want to override is *GetSelection*. *GetSelection* indicates in its Boolean return value whether the document has a portion selected. If there is, the print dialog box gives the option to print only the selected material. The two **var** parameters, *Start* and *Stop*, indicate the position of the selection. *TEditPrintout*, for example, interprets *Start* and *Stop* as character positions, but they can also represent lines of text, pages, and so on.

Printing window contents

The simplest kind of printout to generate is a copy of a window, because windows don't have multiple pages, and window objects already know how to draw themselves on a device context.

To make this common operation even easier, ObjectWindows provides a special kind of printout object, *TWindowPrintout*, to handle it. Any ObjectWindows window object can print its contents to a *TWindowPrintout* object with no modification. The printout object scales the image to fill as much of the page as possible while maintaining the aspect ratio.

Creating a window printout object takes only a single step. All you have to do is construct a window printout object, passing it a title string and a pointer to the window you want printed:

```
PImage := New(PWindowPrintout, Init('Title', PSomeWindow));
```

Often, you'll probably want a window to create a printout of itself, perhaps in response to a menu command:

```
procedure TSomeWindow.CMPrint(var Msg: TMessage);
var P: PPrintout;
```

```
begin
  P := New(PWindowPrintout, Init('Screen dump', @Self));
  { send image to printer }
  Dispose(P, Done);
end;
```

TWindowPrintout doesn't involve paging. When printing a document, you have to print each page individually, but since windows don't have pages, you only have to print one image. And the window already knows how to create that image — it has a *Paint* method. *TWindowPrintout* prints itself by calling your window object's *Paint* method with a printer device context instead of a display context.

Sending printout to a printer

Once you have a printer object and a printout object, whether from a document or from a window, the actual printing is very easy. All you do is call the printer object's *Print* method, passing a pointer to a parent window and a pointer to the printout object:

```
Printer^.Print(PParentWindow, PPrintoutObject);
```

The parent window, in this case, is the window that any pop-up dialog boxes (for printer status and error messages) will be attached to. Normally, this will be the window that generated the print command, such as the main window with the menu bar. During printing, this window is disabled to prevent multiple print commands being sent.

Suppose you have an application whose main window is an instance of *TWidgetWindow*. From that window's menu, you select a menu command to print the contents of the window, generating the command *cm_Print*. The message response method would look like this:

```
procedure TWidgetWindow.CMPrint(var Msg: TMessage);
var P: PPrintout;
begin
  P := New(PWindowPrint, Init('Widgets', @Self));
  Printer^.Print(@Self, P);
  Dispose(P, Done);
end;
```

Choosing a different printer

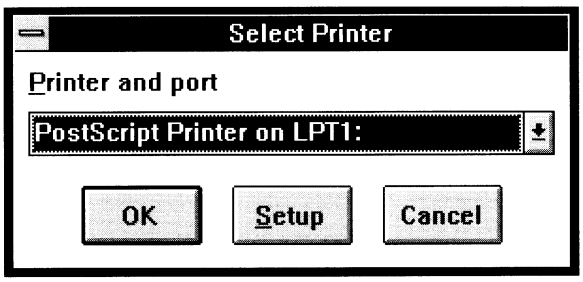
Once you have a printer object in your application, you can associate it with any printer device installed in Windows. By default, *TPrinter* uses the Windows default printer, as specified in the [devices] section of the WIN.INI file.

There are two ways to specify an alternate printer: directly, in code, and through a user dialog box.

Letting the user choose a printer

By far the most common way to assign a different printer is to bring up a dialog box that gives the user the ability to choose from a list of installed printer devices. *TPrinter* does this automatically when you call its *Setup* method. *Setup* uses a *TPrinterSetupDlg* object for this dialog box, as shown in Figure 15.1.

Figure 15.1
The printer setup dialog box



Configuring the printer

One of the buttons in the printer setup dialog allows the user to change the printer's configuration. The Setup button brings up a configuration dialog box defined in the printer's device driver. Your application has no control over the appearance or function of the driver's configuration dialog box.

Assigning a specific printer

In some cases, you might want to assign a specific printer device to your printer object, without user input. *TPrinter* has a *SetDevice* method that does just that.

SetDevice takes three strings as parameters: a device name, a driver name, and a port name.

PART 3

Advanced ObjectWindows

C H A P T E R

16

Windows messages

Windows applications are *event-driven*. That is, rather than proceeding directly from one statement to the next, the flow of control is dictated by outside *events*, such as user interactions and system broadcasts. Applications find out about events they need to respond to by receiving *messages* from Windows.

This chapter describes a number of topics related to sending, receiving, and handling events, including

- What are messages?
- How are messages dispatched?
- What's in a message?
- Handling Windows messages
- Handling commands and notifications
- Defining your own messages
- Sending and posting messages
- Ranges of messages

What is a message?

If you're not used to event-driven programming, Windows might seem like a strange environment. You're probably used to writing programs that spend a lot of their time sitting and waiting for user input through, for example, *Readln* statements.

Event-driven programming gets around that by having the user input handled by a central routine that you don't even have to call. In this case, it's Microsoft Windows itself that interacts with the user and queues up a list of interactions for each running application. These packets of information are called *messages*, and they are simply record structures of type *TMsg*:

```
type
  TMsg = record
    hwnd: HWnd;
    message: Word;
    wParam: Word;
    lParam: Longint;
    time: Longint;
    pt: TPoint;
  end;
```

The fields of the message record give the application information about what sort of event generated the message, where it happened, and what window should react.

One thing to keep in mind about messages is that your application generally gets them *after* something happens. For example, if you resize a window on the screen, the window object gets a *wm_Size* message when you've finished resizing. A few messages are actually requests to take some action. However, in general, they are more notifications that the user or the system has already taken some action to which your code should respond.

Naming messages

The most important field in the message record is *message*, which contains one of the Windows message constants beginning with *wm_*. Each Windows message is uniquely identified by a 16-bit number with a corresponding mnemonic identifier. For instance, a message resulting from a key being pressed has *wm_KeyDown* ($0100) in its *message* field. Messages are usually referred to by their mnemonic names.

Where messages come from

Several different kinds of events can generate messages:

- User interactions, such as keystrokes and mouse clicks or drags
- Windows function calls that need to inform other windows of changes

- Your program explicitly sending a message
- Another application sending a message through DDE (dynamic data exchange)
- Windows itself generating messages, such as for system shutdown

You application generally won't care how messages are generated. The key to event-driven programming is generating and responding to messages. Because Windows and Object-Windows take care of getting messages from one place to another, you can concentrate on producing messages with the proper parameters and responding to messages, rather than on the mechanics of getting messages from one place to another.

Conventional message dispatching

Conventional (that is, nonObjectWindows) Windows applications have a *message loop* to fetch and dispatch messages. Basically, the message loop calls a function associated with the window specified by the window handle in the message record's *hwnd* field.

Every window has a *window function*, specified when the window is created. When Windows finds a message for a particular window, it passes the message to that window's window function. The window function sorts out messages based on the type of message they are, then calls a routine to respond to a particular message.

The program GENERIC.PAS shows the conventional way an application and its windows handle messages. What you find is that every window's window function has a large **case** statement to sort out the messages. This might not seem so bad, until you realize that there are over 100 different messages a window might need to handle. Suddenly the idea of writing and maintaining that **case** statement for every window sounds less appealing!

Even if you have several similar windows, each would probably have to have its own window function with a similar, large **case** statement.

ObjectWindows has a better way

ObjectWindows provides two major improvements over this conventional method of dispatching messages. The first is that the message loop is hidden within the core of your application object. All you have to do to set your application in motion is call your application object's *Run* method. It will then retrieve messages from Windows.

The second major enhancement is automatic message dispatching. Instead of having a window function for each window, you simply define methods in window objects that respond to particular messages. These methods are called *message-response methods*. The ObjectWindows message loop takes care of figuring out which message arrived and calls the appropriate message-response method.

Dynamic virtual methods

The key to automatic message dispatching is an extension to the declaration of virtual methods in objects called *dynamic virtual methods*. In essence, you associate an integer number (such as a message constant) with a method. Your ObjectWindows program (in this case, the application object's message loop) can then call that method based on the message number.

For example, the message generated when you click the left mouse button on a window has *wm_LButtonDown* ($0201) in its *message* field. When the ObjectWindows message loop retrieves such a message for one of its windows, it searches that window object's dynamic method table to see if it has a dynamic method declared for that value. If it finds such a method, it calls it, passing a repackaged message record of type *TMessage* as the parameter. If the window object does not declare a method with that dynamic method index, the message loop calls the window's default window procedure.

Writing message-response methods

To declare a message-response method, you give your window object a procedure named after the message it responds to, with a dynamic method index based on the message constant. For example, to respond to the *wm_LButtonDown* message, you would declare a method as follows:

```
type
  TMyWindow = object(TWindow)
      ⋮
    procedure WMLButtonDown(var Msg: TMessage);
      virtual wm_First + wm_LButtonDown;
      ⋮
  end;
```

The method can actually be called anything you want, but naming response methods after the messages they respond to makes things much clearer. The only actual restrictions on a message-response method are that it must be a procedure, and it must take a single variable parameter of type *TMessage*.

The *wm_First* constant serves as an offset, since ObjectWindows uses several ranges for response methods, and all their indexes are zero-based. By adding an offset to each one, you create a unique dynamic method index. For more information on message ranges, see "Message ranges" in this chapter.

What's in a message?

Now that you know how to create a message-response method, it's time to look at what information is contained in the message. The *TMessage* record that gets passed to your response method looks like this:

```
type
  TMessage = record
    Receiver: HWnd;
    Message: Word;
    case Integer of
      0: (
        WParam: Word;
        LParam: Longint;
        Result: Longint);
      1: (
        WParamLo: Byte;
        WParamHi: Byte;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
  end;
```

The *Receiver* and *Message* fields are not very useful to ObjectWindows objects, since the *Receiver* handle is usually the same as the window object's own *HWindow* field, and *Message* has already been sorted out by the ObjectWindows message loop.

But the other three fields are very important. *WParam* and *LParam* are the 16-bit and 32-bit parameters passed in the original message from Windows. *Result* holds any result code the sender of the message might expect. Note that *TMessage* is a variant record type, so you can access the high-order and low-order bytes or words of the parameters.

The parameter fields

The parameter fields of a message record have different meanings for each message. However, you can make some generalizations.

WParam

The *Word*-type parameter *WParam* usually holds either a handle, an identifier such as a control ID, or a Boolean value. For example, the *wm_SetCursor* message's *WParam* holds the handle of the window containing the cursor. Control notification messages such as *bn_Clicked* have the ID of the affected control in *WParam*. And *wm_Enable* uses *WParam* to hold a Boolean value indicating whether the affected window has been enabled or disabled.

LParam

The *Longint*-type parameter *LParam* usually holds either a pointer value or two *Word*-size values, such as x- and y-coordinates. For example, the *LParam* of *wm_SetText* points to a null-terminated string containing the text being set. Mouse messages, such as *wm_LButtonDown*, use *LParam* to hold the coordinates where the mouse event took place. Using the variant parts of the message record, *LParamLo* holds the x-coordinate, and *LParamHi* the y-coordinate.

The Result field

The *Result* field of the *TMessage* record controls the return value of the message. Sometimes the code that sends a message expects a particular value to come back, such as a Boolean value indicating success or failure or perhaps an error code. You can specify a return value by assigning a value to *Result*.

For example, *wm_QueryOpen* is sent to a window when a user tries to restore it from an iconic state. By default, *wm_QueryOpen*

returns a Boolean *True* (non-zero). If you have a window you want to always be iconic, you could respond to *wm_QueryOpen* and set *Result* to zero, meaning the window cannot be opened:

```
procedure TIconWindow.WMQueryOpen(var Msg: TMessage);
begin
  Msg.Result := 0;
end;
```

Object-oriented message handling

One of the enormous benefits of ObjectWindows' message handling, aside from getting rid of unwieldy **case** statements, is the ability to handle messages in an object-oriented way. That is, your window objects inherit the ability to respond to messages in certain ways, and you only have to change the responses to the particular messages your object will handle differently.

Discarding default behavior

Sometimes when you override the default response to a message, you do so because you don't want the default behavior at all. The simplest case is when you want your object to ignore a message, to not respond at all. To do that, you simply write an empty message-response method. For example, the following method tells an edit control to ignore characters sent in the *wm_Char* message:

```
procedure TNonEdit.WMChar(var Msg: TMessage);
begin
end;
```

Replacing default behavior

A more useful approach than simply ignoring a message might be replacing the default behavior with something entirely different. For example, the following method tells an edit control to beep whenever a key is pressed that would otherwise insert a character:

```
procedure TBeepEdit.WMChar(var Msg: TMessage);
begin
  MessageBeep(0);
end;
```

Beeping when keys are pressed is not particularly useful, of course, but it provides a quick illustration. In general, you can replace the default response to a message by responding in some other way. The response you define will then be the only response.

Augmenting default behavior

On occasion, you might want to *combine* some action with the default response to a message. ObjectWindows provides an object-oriented way to do that. Normally, when you want an object to perform some action based on its inherited behavior, you incorporate a call to the inherited method in your redefined method. ObjectWindows lets you do this for message responses as well.

Calling inherited methods

For example, suppose you've created a new window object, and you want to make it beep when you click the left mouse button on the window in addition to whatever other actions it would normally perform. All you have to do is call the inherited *TWindow.WMLButtonDown* method from your new method:

```
procedure TBeepWindow.WMLButtonDown(var Msg: TMessage);
begin
  inherited WMLButtonDown(Msg);
  MessageBeep(0);
end;
```

In this case, it doesn't make much difference whether you place the inherited *WMLButtonDown* call before or after calling *MessageBeep*. You will have to decide for yourself whether your response methods should call for the inherited actions before or after you do your special processing, based on whether you need or want to alter message parameters.

One thing to keep in mind is that you *can* change the parameters of *Msg* before calling the inherited method. You should do this with care, however, as passing invalid or out-of-range parameters could cause your program to crash Windows, particularly if those parameters contain handles or pointers. If you're careful, changing *Msg* can be useful, but keep in mind that it can also be dangerous!

Calling default procedures

You might have noticed that the section on calling inherited methods used a different example than the preceding sections. You probably expected the new edit control to make a call to its inherited *WMChar* method to handle the default action. That makes sense, but it won't work because *TEdit* doesn't have a *WMChar* method, so an object derived from *TEdit* can't call *WMChar* as an inherited method!

Fortunately, there is still a way for your objects to inherit message-response behavior. When ObjectWindows dispatches a message to an object and that object doesn't define a specific response method, ObjectWindows passes the *TMessage* record to a method called *DefWndProc*, the default window procedure. *DefWndProc* knows how to handle the default actions for all messages. Therefore, if the compiler gives an error when you try to call an inherited message-response method because there is no inherited method for that message, call *DefWndProc* instead.

For example, to add beeping to an edit control *in addition* to inserting typed characters, you need to call *MessageBeep* and *DefWndProc*:

```
procedure TBeepEdit.WMChar(var Msg: TMessage);
begin
  MessageBeep(0);
  DefWndProc(Msg);
end;
```

You can think of *DefWndProc* as taking the place of all inherited message-response methods that aren't specifically defined. Note that this only applies to Windows messages, however. Command messages, notification messages, and control messages all have their own default message handlers. Those messages and their default procedures are all described in the next section.

Commands, notifications, and control IDs

A message is simply a data record identified by a specific value in its *message* field, and ObjectWindows saves you a lot of time and energy by sorting through those messages and dispatching them to message-response methods in your objects. However, the Windows *wm_Command* message has many options you would

still have to sort through with a huge **case** statement, so ObjectWindows handles it for you, too.

For example, the *wm_Command* message is sent whenever a menu item is chosen or an accelerator key is pressed. In a conventional Windows application this message would go to your window function, where a large **case** statement would sort out the different messages, calling some other routine when it found a *wm_Command*. That routine would in turn have to determine *which* command was sent, probably by using another large **case** statement.

Command messages

ObjectWindows handles menu and accelerator commands by dispatching command messages separately, much as it does with other Windows messages. The processing actually takes place inside the *WMCommand* method your window objects inherit from *TWindowsObject*. But instead of handling the commands itself, *WMCommand* dispatches *command messages*, based on the menu or accelerator ID that generated the command.

For example, if you define a menu item with an ID of *cm_DoSomething*, your objects would define response methods based on that ID:

```
type
  TSomeWindow = object(TWindow)
    :
    procedure CMDoSomething(var Msg: TMessage);
      virtual cm_First + cm_DoSomething;
  end;

procedure TSomeWindow.CMDoSomething(var Msg: TMessage);
begin
  { respond to the command }
end;
```

Like *wm_First*, *cm_First* is an ObjectWindows constant defining the beginning of a range of messages. Your command constants must be in the range 0..24,319.

Default command processing

To invoke the default response to a command, you normally call the inherited response method for that command. If the ancestor object doesn't define a response method for the particular command, the default processing is handled by calling *DefCommandProc*. *DefCommandProc* works much like the

DefWndProc method for Windows messages, but it specifically handles commands.

Notification messages

Commands don't have to come from a menu or accelerator. Controls in windows send commands to their parent windows when you click them with the mouse or type in them. These messages are called *notification messages*, and ObjectWindows handles them in two different ways.

Basically, notifications can either go to the control object or to its parent window. If the control has an ObjectWindows object associated with it, ObjectWindows gives the object a chance to respond to the command first. This is called *control notification*. If the control has no associated object, or if the control object does not define a command response, the control's parent window gets a chance to respond. This is called *parent notification*.

Control notifications

Normally, controls don't need to do anything special in response to user actions; the default is the expected behavior. But if you want the control to do something extra or something different, notification messages give your objects a chance to do that.

For example, suppose you want a button to beep every time it is clicked. You simply give your button object a method to respond to the notification that it's been clicked:

Listing 16.1
Defining a control-notification response

```
type
  TBeepButton = object(TButton)
    procedure BNClicked(var Msg: TMessage);
      virtual nf_First + bn_Clicked;
  end;

procedure TBeepButton.BNClicked(var Msg: TMessage);
begin
  MessageBeep(0);
end;
```

ObjectWindows defines the *nf_First* constant as an offset defining a range of messages used for notifying controls.

Parent notification

If a control has no interface object associated with it, or if the control object does not define a response to a particular command, notification messages are sent to the control's parent window object instead of to the control object. Because the parent window needs to know *which* of its controls caused the notification, the notification message to the parent is based on the ID of the control.

For example, to respond to notifications of interactions with a control with the ID *id_MyControl*, you define a method as follows:

```
type
  TMyWindow = object(TWindow)
      :
    procedure IDMyControl(var Msg: TMessage);
      virtual id_First + id_MyControl;
  end;

procedure TMyWindow.IDMyControl(var Msg: TMessage);
begin
  { perform the response }
end;
```

ObjectWindows defines the constant *id_First* as an offset to the range of messages used to respond to control notifications.

Windows rarely have a default behavior defined in response to particular controls. However, if you want to make sure default behavior is taken care of, you can call *DefChildProc*. *DefChildProc* works much like *DefWndProc*, but handles parent notification messages from controls instead of Windows messages.

Notifying controls and parents

It is possible that you will sometimes need to have both a control *and* its parent window respond to some user interaction with the control. Again, ObjectWindows provides a way to do this. Since parent notification is the default response when a control object does not define a notification response, all you have to do is invoke the default response in addition to defining a control response.

For example, given the *TBeepButton* example in Listing 16.1, you can also notify the parent window by adding a call to *DefNotificationProc*:

```
procedure TBeepButton.BNClicked(var Msg: TMessage);
begin
  MessageBeep(0);
  DefNotificationProc(Msg);
end;
```

Calling *DefNotificationProc* ensures that the control's parent window receives an ID-based notification message, just as if the control had defined no response at all.

Defining your own messages

Windows reserves 1,024 messages for its own internal use. All the standard message fall in that range. The top of the message range is defined by a constant, *wm_User*. To define a message for use by the windows in your application, define a message ID that falls in the range *wm_User..wm_User*+31,744.

To avoid conflicts with standard messages, you should define your message IDs as constants based on *wm_User*. For example, in an application where you need three user-defined messages, you might declare them as follows:

```
const
  wm_MyFirstMessage = wm_User;
  wm_MySecondMessage = wm_User + 1;
  wm_MyThirdMessage = wm_User + 2;
```

Responding to your messages is just like responding to any other message:

```
TCustomWindow = object(TWindow)
   :
  procedure WMMyFirstMessage(var Msg: TMessage);
    virtual wm_First + wm_MyFirstMessage
```

As a general rule, user-defined messages should only be used for internal messaging. If you send a user-defined message to another application, you need to be absolutely sure that other application defines the message the same way your application does. External communication is best handled using dynamic data exchange (DDE).

Sending messages

So far, this chapter has discussed responding to messages, but hasn't really talked about how to generate messages. Most Windows messages are generated by Windows itself, in response to user actions or system events. But your program can generate messages, either to simulate user actions, or to manipulate elements on the screen.

ObjectWindows already provides ways for you to send many of the messages you might otherwise have to send manually. For example, a common use for generating messages is manipulation of controls. Windows defines messages such as *lb_AddString* to add a string to a listbox and *bm_SetCheck* to check or uncheck a check box or radio button. ObjectWindows defines methods for control objects (*TListBox.AddString* and *TCheckBox.SetCheck*) that send these messages for you, without your having to even think about the need to use messages.

You will probably find that object methods already exist to handle most of the functions you might otherwise perform by sending messages.

Sending and posting messages

However, there will probably be times when you want to send messages to windows in your applications. Windows provides two functions that enable you to generate messages: *SendMessage* and *PostMessage*. The primary difference between the two functions is that *SendMessage* waits for the message to be processed before returning. *PostMessage* just adds the message to the message queue and returns.

Keep in mind that *SendMessage*, especially when called from within a message-response method, can cause infinite loops or "race" conditions to occur that will crash your programs. Not only should you avoid the obvious loops such as a message-response method that generates the very message that called it, but also avoid sending messages with side-effects. For example, an object's *Paint* method, which gets called in response to *wm_Paint* messages, obviously shouldn't explicitly send another *wm_Paint* message to itself. But it should also avoid performing other actions that would result in another *wm_Paint* message being sent while the method is still active, such as resizing the

window, invalidating the window, or creating or destroying overlapping windows.

Sending a message

Sending a message takes four pieces of information: the handle of the receiving window, the message number, and the *Word* and *Longint* parameters. In an ObjectWindows application, the recipient's handle is usually the *HWindow* field of the interface object. The message ID is just the constant that identifies the particular message you want to send, such as *wm_Move* or *em_SetTabStops*. The parameters vary depending on the message.

The value returned by *SendMessage* is the value of the *Result* field in the message record when processing is completed. Keep in mind that if you call an inherited or default message-response method after you set the value of *Result*, your value might be overwritten.

Windows provides a limited message-broadcasting facility with *SendMessage*. If you pass $FFFF as the handle of the window you want to send to, Windows sends the message to all pop-up and overlapped windows in the system — not just in your application. You should therefore never broadcast a user-defined message, since you can't be sure another application hasn't defined that message to mean something entirely different.

Posting a message

Posting a message is useful if you don't particularly care *when* the message is responded to, or if you need to avoid the loops that *SendMessage* can cause. If all you want to do is make sure a message is sent, and you can proceed without waiting for the response, you can call *PostMessage*. It's parameters are identical to those for *SendMessage*. *PostMessage* should not be used to send messages to controls.

Sending a message to a control

Probably the only time you'll ever need to send a message to an element on the screen that doesn't have an associated object is in the case where you need to communicate with a control in a dialog box. Again, the *easiest* way to deal with this is to use objects, either for the control or for the dialog box or both.

If the control has an object associated with it, you can just use the object's *HWindow* field to get the control's handle. If the dialog box has an object associated with it, you can call the dialog box object's *SendDlgItemMsg* method, which takes the control ID, the message ID, and the two message parameters. In most ObjectWindows

applications, either of these approaches should be available to you.

If for some reason you have neither a dialog box object or a control object available, you can send a message to a control in a dialog box using the API function *SendDlgItemMessage*, which takes as its parameters the dialog box handle, the control ID, the message ID, and the two message parameters.

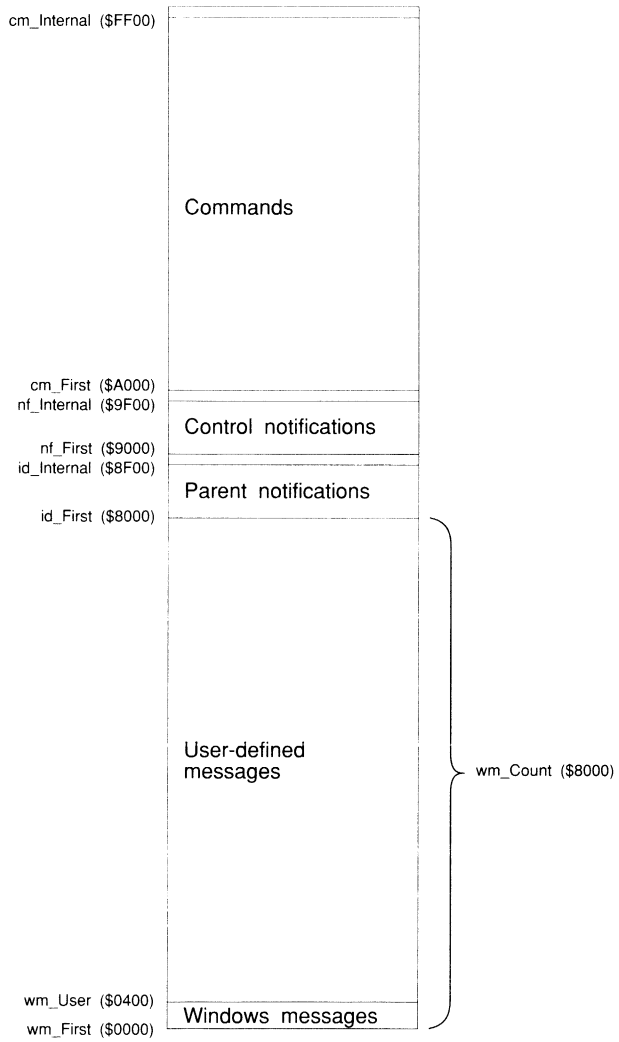
Message ranges

A message is defined by the *message* field of the message record, so it's a 16-bit value. Windows reserves messages 0..$03FF for its own standard messages, and the rest of the messages up to $7FFF for user-defined messages. ObjectWindows divides the remaining messages into ranges for commands and notifications, as shown in Table 16.1.

Table 16.1
Message ranges

Range	Values
Reserved Windows messages	$0000-$03FF
User-definable messages	$0400-$7FFF
Control-notification messages	$8000-$8EFF
ObjectWindows reserved control notifications	$8F00-$8FFF
Parent-notification messages	$9000-$9EFF
ObjectWindows reserved parent notifications	$9F00-$9FFF
Command messages	$A000-$FEFF
ObjectWindows reserved commands	$FF00-$FFFF

Figure 16.1
Message and command ranges



C H A P T E R

17

The Graphics Device Interface

Many types of Windows applications need only windows, dialog boxes, and controls for a full-featured user interface. But some applications, such as drawing and image manipulation applications, require graphics to fill their windows. These graphics can be in the form of lines, shapes, text, and bitmapped images.

Windows has a set of functions collectively called the Graphics Device Interface, or GDI. You can think of GDI as the engine your Windows applications use to display and manipulate graphics. GDI functions give your application drawing capabilities independent of the display device used. For example, you could use the same functions that you use to write to an EGA display as to write to a VGA display or even to a PostScript printer.

GDI achieves this device independence by using device drivers to translate your GDI function calls into commands meaningful to the output device being used. This means you don't have to worry about *how* a particular output device renders an image. You tell the driver to do something, and it manipulates the device as needed.

Writing to an output device

Unlike traditional DOS-based graphics programs, Windows programs never directly write to screen pixels or printers, but

rather to a logical entity called a *device context*. A device context is a virtual surface with associated attributes, such as a pen, brush, font, background color, text color, and current position. To your application, all device contexts look alike, no matter what the device is actually like.

When you call GDI functions to draw on a device context, the device driver associated with that device context translates that drawing action into appropriate commands. These commands reproduce the drawing action as accurately as possible on the device, regardless of the device's capabilities. The display might be a low-resolution monochrome screen, a four-million-color screen, or a graphics plotter.

Remember that the device context represents only that part of the device you can actually draw on. Although you might think of your output in terms of an entire window (frame, menu, client area) or printed page, the device context only covers the part where you draw: the client area of a window or the printable part of a page.

A device context is a Windows-managed element, much like a window element, except that a device context has no corresponding ObjectWindows object.

How is a display context different?

Windows provides special device contexts for the client areas of windows, called *display contexts*. Rather than representing a device *per se*, such as a screen or printer, the display context lets you treat the client area of a window as an entire device by itself.

In practice, you don't treat a display context differently than you would any other device context. What it allows you to do is act as if your window is an entire device, so you don't have to worry about offsets for screen position and so on.

Managing display contexts

In order to draw graphics onto a display context, your program must first obtain a display context for the desired window. Because the memory requirement of a display context is great, however, only five display contexts are concurrently accessible during each Windows session. This means that each window cannot maintain its own display context. It must obtain one only when it is needed, and release it as soon as possible. This may

seem disconcerting because you do not have control of the display context attributes; another window, or even another application, could change the display context's attributes. In addition, drawing tools are not part of a display context's memory. They must be selected into the display context each time it is obtained.

Managing a display context

Typically, you will define a window object field to store the handle to the current display context, much as *HWindow* stores a handle to a window:

```
type
  TMyWindow = object(TWindow)
    TheDC: HDC;
      ⋮
  end;
```

To obtain a display context for a window, call the Windows function *GetDC*:

```
TheDC := GetDC(HWindow);
```

Then you can perform drawing operations on the display context. You can use the display context handle in Windows graphics functions:

```
LineTo(TheDC, Msg.LParamLo, Msg.LParamHi);
```

As soon as you are done with the display context, release it with a call to the *ReleaseDC* function:

```
ReleaseDC(HWindow, TheDC);
```

Do not call *GetDC* twice in a row, with no *ReleaseDC* call in between. This will eventually lead to a system failure while your program is running because you will run out of available display contexts.

What's in a device context?

A device context holds a set of *drawing tools*, such as pens, brushes, and fonts. Technically, these tools don't really exist. They are just convenient ways of thinking about certain groups of drawing properties.

A pen, for example, is just a convenient way of thinking about the line-drawing properties of a device. A graphical line between two points has three properties: width, color, and style (such as dashed or dotted), which you can treat together as a "pen." These logical groupings of drawing properties make Windows graphics much simpler than if you had to consider them all equally.

You normally don't need to alter most of the attributes of a device context, but it is important to know just what the device context contains. This section briefly describes the properties of device contexts, including bitmaps, colors, mapping, clipping, and drawing tools.

Bitmapped graphics

The actual surface of the display context is called a *bitmap*. Bitmaps represent the memory configuration of a particular device. Thus, they are dependent on the kind of device being addressed. This poses a problem, because bitmaps saved from one device might be incompatible with another device. GDI provides some techniques to address this problem, including *device-independent bitmaps*. GDI functions that create bitmaps include *CreateCompatibleDC*, *CreateCompatibleBitmap*, and *CreateDIBitmap*. GDI functions to manipulate bitmaps include *BitBlt*, *StretchBlt*, *StretchDIBits*, and *SetDIBitsToDevice*.

Drawing tools

To perform most of the actual drawing, the display context uses three tools: a *pen*, a *brush*, and a *font* tool.

Pens

The pen is used to draw lines, arcs, and polylines, which are multiple line segments. The attributes of a pen include its color, width, and style (solid or dotted, for example).

Brushes

A brush is used when filling solid shapes, such as rectangles, rounded rectangles, ellipses, and polygons. Functions that draw solid shapes use the pen to draw edges and the brush to fill the interiors. There are four types of brushes: solid, hatched, bitmap patterned, and device-independent bitmap patterned.

Fonts

The font tool is used when drawing text on the display context. It specifies the font's height, weight, pitch, family, and face name.

All three tools use the display context's *background color* attribute to fill in spaces. The drawing tools are covered in depth in the "Drawing tools" section of this chapter.

Color

The colors that a device uses for drawing are held in a *color palette*. If a color you want to use is not available in the color palette, you can add it. More commonly, you allow the device driver to approximate the desired color by dithering the palette colors. Color palettes are explained in greater detail in the "Using palettes" section of this chapter.

Mapping modes

It is difficult to choose units for drawing when you don't know what device will be used for display. Most applications ignore this problem and assume the default unit (one pixel) will work well enough. However, some applications demand that the display exactly reproduce the dimensions of the desired image. For such applications, GDI allows different *mapping modes*, some of which are device independent. Each mapping mode has a unit and a coordinate orientation. The default mapping mode sets the origin to the upper left corner of the display context with the positive x-axis pointing right and the positive y-axis pointing down. Every display context has a mapping mode attribute to determine how to interpret the coordinates you give it.

Sometimes it is necessary to translate between the logical coordinates you use to draw with and the physical bitmap coordinates the current mapping mode translates them to. For most applications, the origin for the screen is its upper-left corner, but for a window, the origin is the upper-left corner of the window's client area. Some windows scroll their client surface so that the origin is not even in the client area. Some GDI functions operate in particular coordinate systems, so conversions are necessary. GDI provides functions to handle these coordinate translations, such as *ScreenToClient*, *ClientToScreen*, *DPToLP*, and *LPtoDP*.

Clipping regions

To prevent drawing outside of the intended area, every display context has a *clipping region* attribute. A clipping region can be a complex polygonal or elliptical shape inside of which any drawing on the display context's virtual surface can actually appear. For most applications, the default clipping region, the window's client area, will suffice. Only applications that produce special visual effects need to alter the clipping region.

Using Windows' drawing tools

The display context manages the display of graphics on the screen. To display graphics in different ways, you can modify the drawing tools with which you render the graphics.

To assign the attributes of a drawing tool, a Windows program selects either a *stock tool* or a *logical tool* into a device context. A stock tool is an existing, Windows-defined tool representing the most common attribute choices, such as a solid black pen, a gray brush, or the system font. A logical tool is one your program creates by filling the fields of a particular record, a *TLogPen*, *TLogBrush*, or *TLogFont*.

Stock tools

Stock tools are created with the GDI function *GetStockObject* with a parameter specifying which stock tool you use. For example:

```
var TheBrush: HBrush
begin
  TheBrush := GetStockObject(LtGray_Brush);
   :
end;
```

where *LtGray_Brush* is an integer constant defined in the *WinTypes* unit. Table 17.1 lists all the stock tool constants:

Table 17.1
Stock drawing tools

Brushes	Pens	Fonts
White_Brush	*White_Pen*	*OEM_Fixed_Font*
LtGray_Brush	*Black_Pen*	*ANSI_Fixed_Font*
Gray_Brush	*Null_Pen*	*ANSI_Var_Font*
DkGray_Brush		*System_Font*

Table 17.1: Stock drawing tools (continued)

Black_Brush	*Device_Default_Font*
Null_Brush	*System_Fixed_Font*
Hollow_Brush	

Unlike logical tools, stock tools should *not* be deleted after use.

Logical tools

The logical tool records, *TLogPen*, *TLogBrush*, and *TLogFont*, contain fields to hold each tool attribute. For example, *TLogPen.lopnColor* holds the pen's color value. Each record type defines its own set of attributes, appropriate to the type of tool.

Logical pens

You can create logical pens with the Windows functions *CreatePen* or *CreatePenIndirect*. For example:

```
ThePen := CreatePen(ps_Dot, 3, RGB(0, 0, 210));
ThePen := CreatePenIndirect(@ALogPen);
```

Here is the *TLogPen* record definition:

```
TLogPen = record
  lopnStyle: Word;
  lopnWidth: TPoint;
  lopnColor: Longint;
end;
```

The style field, *lopnStyle*, holds a constant indicating the line style.

Figure 17.1
Line styles for pen tools

Constant	Result
PS_SOLID	
PS_DASH	
PS_DOT	
PS_DASHDOT	
PS_DASHDOTDOT	
PS_NULL	

The width field, *lopnWidth*, holds a point whose x-coordinate is an integer indicating the line width in device coordinates. On a VGA screen, if the value is zero, a line with width 1 pixel is drawn. The y-coordinate value is ignored.

The color field, *lopnColor*, holds a *Longint* whose bytes represent the intensity values for the primary colors red, green and blue, which can be combined to produce any color. The *lopnColor* value must be in the form $00*bbggrr*, where *bb* is a placeholder for the blue value, *gg* for the green value, and *rr* for the red value. The acceptable range of intensity for each primary color is 0 to 255, or 0 to FF in hexadecimal. The following table shows some sample color values.

Table 17.2
Sample RGB color values

Value	Color
$00000000	black
$00FFFFFF	white
$000000FF	red
$0000FF00	green
$00FF0000	blue
$00808080	gray

As an alternative, you can use the *RGB* function to produce colors. *RGB(0, 0, 0)* returns black, *RGB(255, 0, 0)* returns red, and so on.

Logical brushes

You can create logical brushes with the Windows functions *CreateHatchBrush*, *CreatePatternBrush*, *CreateDIBPatternBrush*, or *CreateBrushIndirect*. For example:

```
TheBrush := CreateHatchBrush(hs_Vertical, RGB(0, 255, 0));
TheBrush := CreateBrushIndirect(@ALogBrush);
```

Here is the *TLogBrush* record definition:

```
TLogBrush = record
  lbStyle: Word;
  lbColor: Longint;
  lbHatch: Integer;
end;
```

The *lbStyle* field holds an integer constant indicating the brush's style:

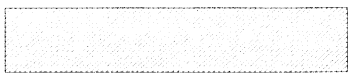
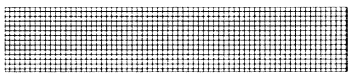
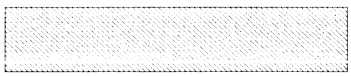
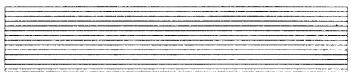
- *bs_DIBPattern* indicates that the pattern brush is defined by a device-independent bitmap.
- *bs_Hatched* specifies one of the predefined hatched patterns (see *lbHatch*).
- *bs_Hollow* is a null brush.

- *bs_Pattern* uses the 8-by-8 pixel top-left corner of a bitmap that is currently in memory.
- *bs_Solid* is a solid brush.

The *lbColor* field holds a color value like the one required for *TLogPen* records. This field is ignored by brushes with styles *bs_Hollow* or *bs_Pattern*.

The *lbHatch* field holds an integer constant indicating the hatched pattern for brushes with the style *bs_Hatched*. If the style is *bs_DIBPattern*, *lbHatch* holds a handle to the bitmap.

Figure 17.2
Hatch styles for brush tools

Constant	Result
HS_BDIAGONAL	
HS_CROSS	
HS_DIAGCROSS	
HS_FDIAGONAL	
HS_HORIZONTAL	
HS_VERTICAL	

Logical fonts

You can create logical fonts using the Windows functions *CreateFont* or *CreateFontIndirect*.

Here is the *TLogFont* record definition:

```
TLogFont = record
  lfHeight: Integer;
  lfWidth: Integer;
  lfEscapement: Integer;
  lfOrientation: Integer;
  lfWeight: Integer;
  lfItalic: Byte;
  lfUnderline: Byte;
  lfStrikeOut: Byte;
```

```
    lfCharSet: Byte;
    lfOutPrecision: Byte;
    lfClipPrecision: Byte;
    lfQuality: Byte;
    lfPitchAndFamily: Byte;
    lfFaceName: array[0..lf_FaceSize - 1] of Byte;
  end;
```

When you use a *TLogFont* to create a font tool, you are specifying attributes of the desired font. However, your program does not take this information and generate a screen font from available information. Instead, it maps the font request to a screen font that is currently defined in the Windows session.

The *lfHeight* field specifies the desired height of the font. A zero value results in a default size. A positive value is the requested cell height in logical units. A negative value is made positive and is the requested character height in logical units.

The *lfWidth* field provides the desired width in device units. If zero, the aspect ratio is preserved.

For rotated text, set *lfEscapement* to a value, in tenths of degrees, by which the text is rotated counterclockwise. The *lfOrientation* does the same rotation for each character.

The desired weight is specified in *lfWeight*. You can use font weight constants, such as *fw_Light*, *fw_Normal*, *fw_Bold*, and *fw_DontCare*.

Three attributes of a font—italic, underline, and strike out—are requested with non-zero values in *lfItalic*, *lfUnderline*, and *lfStrikeOut*.

The *lfCharSet* field requests a particular character set, *ANSI_CharSet*, *OEM_CharSet*, or *Symbol_CharSet*. The ANSI character set is listed in Appendix B of the *Microsoft Windows User's Guide*. *OEM_CharSet* is system-dependent.

The field *lfOutPrecision* specifies how precisely the font Windows provides must match the size and position requests. The default is *Out_Default_Precis*. The field *lfClipPrecision* specifies how to clip partially visible characters. The default is *Clip_Default_Precis*.

The *lfQuality* field indicates how closely the font Windows provides matches the requested font attributes. It can be set to *Default_Quality*, *Draft_Quality*, or *Proof_Quality*. With *Proof_Quality*, bold, italic, underline and strikeout fonts are synthesized if not available.

The field *lfPitchAndFamily* requests a pitch and font family. It can be the result of an **or** operation between one pitch constant and one family constant.

Table 17.3
Font pitch and family constants

Pitch Constants	Family Constants
Default_Pitch	*ff_Modern*
Fixed_Pitch	*ff_Roman*
Variable_Pitch	*ff_Script*
	ff_Swiss
	ff_Decorative
	ff_DontCare

Finally, *lfFaceName* is a string that specifies the requested typeface. If its value is 0, you will get a typeface based on the values in the other *TLogFont* fields.

Here are a few sample fonts with the source code that defined their *TLogFont* records:

Sample Text

```
procedure MyWindow.MakeFont;
var MyLogFont: TLogFont;
begin
  with MyLogFont do
  begin
    lfHeight := 30;
    lfWidth := 0;
    lfEscapement := 0;
    lfOrientation := 0;
    lfWeight := fw_Bold;
    lfItalic := 0;
    lfUnderline := 0;
    lfStrikeOut := 0;
    lfCharSet := ANSI_CharSet;
    lfOutPrecision := Out_Default_Precis;
    lfClipPrecision := Clip_Default_Precis;
    lfQuality := Default_Quality;
    lfPitchAndFamily := Variable_Pitch or ff_Swiss;
    StrCopy(@lfFaceName, 'Helv');
  end;
  TheFont := CreateFontIndirect(@MyLogFont);
end;
```

Sample Text

```
procedure MyWindow.MakeFont;
var MyLogFont: TLogFont;
```

```
begin
  with MyLogFont do
  begin
    lfHeight := 10;
    lfWidth := 0;
    lfEscapement := 0;
    lfOrientation := 0;
    lfWeight := fw_Normal;
    lfItalic := Ord(True);
    lfUnderline := Ord(True);
    lfStrikeOut := 0;
    lfCharSet := ANSI_CharSet;
    lfOutPrecision := Out_Default_Precis;
    lfClipPrecision := Clip_Default_Precis;
    lfQuality := Default_Quality;
    lfPitchAndFamily := Fixed_Pitch or ff_DontCare;
    StrCopy(@lfFaceName, 'Courier');
  end;
  TheFont := CreateFontIndirect(@MyLogFont);
end;
```

Σαμπλε Τεξτ

```
procedure MyWindow.MakeFont;
var MyLogFont: TLogFont;
begin
  with MyLogFont do
  begin
    lfHeight := 30;
    lfWidth := 0;
    lfEscapement := 0;
    lfOrientation := 0;
    lfWeight := fw_Normal;
    lfItalic := 0;
    lfUnderline := 0;
    lfStrikeOut := 0;
    lfCharSet := Symbol_CharSet;
    lfOutPrecision := Out_Default_Precis;
    lfClipPrecision := Clip_Default_Precis;
    lfQuality := Proof_Quality;
    lfPitchAndFamily := Variable_Pitch or ff_Roman;
    StrCopy(@lfFaceName, 'Tms Rmn');
  end;
  TheFont := CreateFontIndirect(@MyLogFont);
end;
```

Using drawing tools

Besides allowing you to draw on a window, a display context holds drawing tools: the pens, brushes, fonts, and palettes you use to draw text and graphics. When you draw a line on a display context, the line appears with the attributes of the current pen, such as its color, its style (solid, dotted, and so on) and it thickness. When you fill in a region, it appears with the attributes of the current brush, such as its fill pattern and its color. When you draw text in a display context, it appears with the font (Modern, Roman, Swiss, and so on), size, and style (italic, bold, and so on) of the current font tool. A palette holds a collection of the current, available colors.

A display context holds one of each type of drawing tool. A newly obtained display context holds default tools: a thin black pen, a solid black brush, a system font, and a default palette. If you are satisfied with these tools, you will never need to modify them.

To change these default tools, you must create a new tool element and *select* it into a display context. When you select a new pen, for example, the old pen is automatically deselected. We recommend that you save the previous tools and reselect them when you are done using the new tool:

```
var
  NewPen, OldPen: HPen;
  TheDC: HDC;
begin
  { create a pen with a thickness of 10 }
  NewPen := CreatePen(ps_Solid, 10, RGB(0, 0, 0));
  TheDC := GetDC(AWindow^.HWindow);
  OldPen := SelectObject(TheDC, NewPen);
  { perform drawing }
  SelectObject(TheDC, OldPen);
  ReleaseDC(AWindow^.HWindow, TheDC);
  DeleteObject(NewPen);
end;
```

As this example shows, new drawing tools must be created and then deleted. Like display contexts, they are elements stored in Windows memory. Failure to delete them results in memory loss and eventual failure. Also like display contexts, you should store handles to drawing tools in variables of type *HPen*, *HBrush*, *HFont*, or *HPalette*.

The Windows function *DeleteObject* removes drawing tools from Windows memory. *Do not* delete drawing tools that are currently selected into a display context!

While a display context can only have one of each type of drawing tool selected at a time, you can keep a stable of available drawing tools. The important thing is to delete them all before your application terminates. One approach, used in the example in Chapter 2, is to define a window object field called *ThePen* to store a handle to the current pen tool. When your user selects a new pen style, a new pen tool is created and the old one is deleted. Then, the final pen is deleted in the main window's *CanClose* method. You need not delete the default tools supplied with a newly obtained display context.

There are two ways to create new drawing tools. The easiest approach is to use an existing, alternative tool called a *stock* tool. The stock tools are listed in Table 17.1.

To set a stock tool into a display context object, use the methods *SetStockPen*, *SetStockBrush*, *SetStockFont*, and *SetStockPalette*. For example,

```
ThePen := GetStockObject(Black_Pen);
```

Do not delete stock tools from Windows memory, as you would custom tools.

Sometimes, there is no stock tool that has the attributes you desire. For example, all the stock pens produce thin lines, and you might want heavy lines. In that case, there are two ways to create custom drawing tools. One way is to call the Windows functions *CreatePen*, *CreateFont*, *CreateSolidBrush*, or *CreateDIBPatternBrush*. These functions take parameters that describe the desired tool and return tool handles which can be used in *SelectObject* calls.

Another way to create custom tools is to build a description of the attributes of a tool called a *logical* tool. A logical tool is embodied by the Windows data structures *TLogPen*, *TLogBrush*, *TLogFont*, and *TLogPalette*. For example, a *TLogPen* has fields to hold the thickness, the color, and the style. Once you have set up a logical tool data structure, pass it as a parameter to *CreatePenIndirect*, *CreateBrushIndirect*, *CreateFontIndirect*, or *CreatePalette*. These functions return tool handles that can be used in *SelectObject* calls. This example sets the pen of the window's display context to be blue:

```
procedure SampleWindow.ChangePenToBlue;
var
  ALogPen: TLogPen;
  ThePen: HPen;
begin
  ALogPen.lopnColor := RGB(0, 0, 255);
  ALogPen.lopnStyle := ps_Solid;
  ALogPen.lopnWidth.X := 0;
  ALogPen.lopnWidth.Y := 0;
  ThePen := CreatePenIndirect(@ALogPen);
  SelectObject(TheDC, ThePen);
end;
```

Displaying graphics in windows

Painting is the process of displaying the contents of a window. A Windows application is responsible for painting its windows when they are first displayed and when they need updating, for example, after being restored from an icon or uncovered by another window. While Windows does not provide automatic painting of a window's contents, it does notify the window when it needs to paint itself. This section shows how to draw in a window, describes the painting mechanism, and explains the use of display contexts.

In this section, *painting* and *drawing* both refer to displaying graphics in a window. *Painting* refers to the automatic display of graphics when the window first appears or needs updating. *Drawing* refers to creating and displaying a specific graphic at any other time, under direct program control. *Graphics* refers to both text and graphic elements, such as bitmaps and rectangles.

Painting windows

When a window is in need of painting, it has been *invalidated*, meaning its display is no longer valid and needs to be updated. This happens as a result of the window being initially displayed, restored after being minimized to an icon, or after another window is moved, revealing part of the window behind it. In all of these cases, Windows sends a *wm_Paint* message to the appropriate application. This message automatically results in a call to your window's *Paint* method. One of *Paint*'s parameters, *PaintDC*, is the display context to be used for painting.

TWindow's *Paint* method does no painting, since *TWindow* objects have no graphics to paint. In your window types, define a *Paint* method that calls methods and functions that draw graphics and text in the window.

One thing you can do with a display context is to select into it new drawing tools, such as pens of different colors, or brushes of different patterns. You will have to reselect these tools into the paint display context in your *Paint* method.

At the conclusion of the *Paint* method, the paint display context is automatically released.

Graphics strategy

The *Paint* method is responsible for drawing the current contents of a window at any time, including the window's first appearance. Thus, the *Paint* method should be capable of drawing all of the window's "permanent" graphics. In addition, it should be able to recreate any graphics added to the window since its initial appearance. In order to produce these "dynamic" graphics, the *Paint* method must have access to the instructions or data by which the graphics were produced.

You can choose from two approaches. The first is to isolate your graphics code in only a few methods, and to call these methods when dynamically producing graphics, *and* from the *Paint* method. The other approach, shown in the example in Chapter 2, is to store data relating to the graphics content of the window in a field of that window's object. This data can include coordinates, formulas, and bitmaps, for example. Then, in the *Paint* method, replay the graphics routines required to transform this data into graphics.

Using these strategies and the ability of the object to store its own data and functions, you can produce live graphics applications that never skip a beat.

Drawing in windows

In order to draw any text or graphics in a window object, you must *obtain* a display context. After drawing, you must *release* the display context. (There are only five display context elements available in any one Windows session.) In between that time, you can use the display context handle as an argument in any Windows graphics function.

Calling windows graphics functions

One of the rules of GDI is that its functions require a handle to a display context as an argument to work. Usually, you will call these functions from within the methods of a window type. For example, *TextOut* is a function that draws text on a display context at the specified location:

```
TheDC := GetDC(HWindow);
TextOut(TheDC, 50, 50, 'Sample Text', 11);
ReleaseDC(HWindow, TheDC);
```

GDI drawing functions

This section describes the various API calls that you can use to draw things in windows.

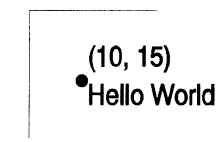
Drawing text

Text drawing functions use the specified display context's current font to draw. The *TextOut* function draws text at a specified point. Depending on the value of the current text formatting flags, *TextOut* aligns the text. The default is left aligned. The current alignment can be retrieved with the *GetTextAlign* function and set with the *SetTextAlign* function.

The *TextOut* function is the most often used text-drawing function. Using the default text-formatting flag settings, the following *Paint* method draws a left-justified character array of which the upper-left corner is at (10, 15).

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
  MyTextString: array[0..20] of Char;
begin
  StrCopy(MyTextString, 'Hello, World');
  TextOut(PaintDC, 10, 15, MyTextString, StrLen(MyTextString));
end;
```

Figure 17.3
The results of the TextOut function



Drawing lines

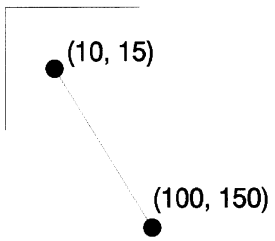
Line drawing functions use the specified display context's current pen to draw. Most lines are drawn using the *MoveTo* and *LineTo* functions. These functions affect a display context attribute called the *current position*. To use the analogy of drawing with a pencil and paper, the current position is the point where the pencil touches the paper.

MoveTo and LineTo

The *MoveTo* function moves the current position to the specified coordinate. The *LineTo* function draws a line using the pen from the current position to the specified coordinate. It then updates the current position to the specified coordinate. The following *Paint* method draws a line from (100, 150) to (10, 15):

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
begin
  MoveTo(PaintDC, 100, 150);
  LineTo(PaintDC, 10, 15);
end;
```

Figure 17.4
The results of the LineTo function



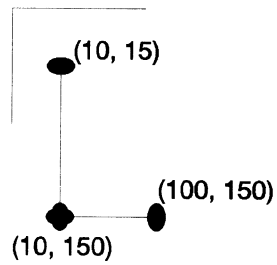
PolyLine

The *Polyline* function draws a series of lines in a connect-the-dots fashion. This is similar to using an initial *MoveTo* and subsequent *LineTo* functions; however, *Polyline* performs the operation much more quickly and does not affect the current position of the pen. The following *Paint* method draws a right angle.

```
procedure TMyWindow.Paint (PaintDC: HDC; var PaintInfo:
TPaintStruct);
var Points: array[0..2] of TPoint;
begin
  Points[0].X := 10;
  Points[0].Y := 15;
  Points[1].X := 10;
  Points[1].Y := 150;
  Points[2].X := 100;
```

```
  Points[2].Y := 150;
  Polyline(PaintDC, @Points, 3);
end;
```

Figure 17.5
The results of the Polyline function



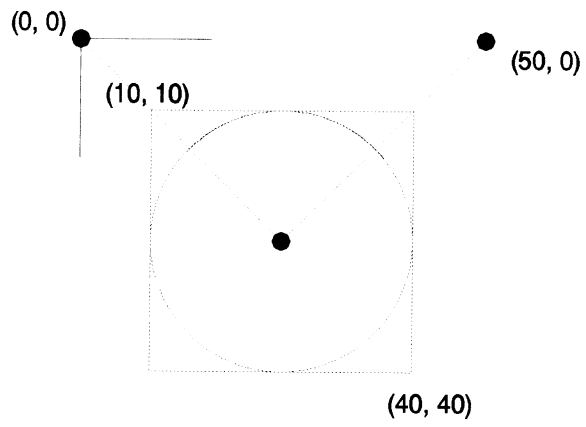
Arc

The *Arc* function draws an arc along the perimeter of the ellipse bounded by the specified rectangle. The arc starts at the intersection of the ellipse edge and the line from the center of the ellipse to the specified starting point. The arc is drawn counterclockwise until it reaches the position where the ellipse edge intersects the line from the center of the ellipse to the specified ending point.

The following *Paint* method draws the top quarter of a circle starting at (40, 25) and ending at (10, 25) using the bounding rectangle (10, 10), (40, 40), the starting point (0, 0), and the ending point (50, 0). This happens even though the specified starting and ending points are not on the arc.

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
begin
  Arc(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
end;
```

Figure 17.6
The results of the Arc function



Drawing shapes

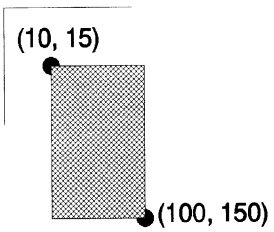
Shape drawing functions use the specified display context's current pen to draw the perimeter and the current brush to fill the interior. They do not affect the current position.

Rectangle

The *Rectangle* function draws a rectangle from the upper left corner to the lower right corner. For example, the following statement in a *Paint* method draws a rectangle from (10, 15) to (100, 150).

```
Rectangle(PaintDC, 10, 15, 100, 150);
```

Figure 17.7
The results of the Rectangle function

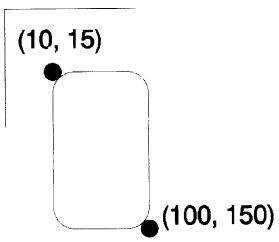


RoundRect

The *RoundRect* function draws a rectangle with rounded corners. The rounded corners are defined as quarters of an ellipse. For example, the following statement in a *Paint* method draws a rectangle from (10, 15) to (100, 150), the corners of which will be quarters of an ellipse with a width of 9 and a height of 11.

```
RoundRect(PaintDC, 10, 15, 100, 150, 9, 11);
```

Figure 17.8
The results of the RoundRect function

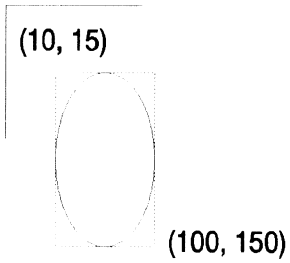


Ellipse

The *Ellipse* function draws an ellipse defined by a bounding rectangle. The following example draws an ellipse within the rectangle (10, 15) to (110, 70).

```
Ellipse(PaintDC, 10, 50, 100, 150);
```

Figure 17.9
The results of the Ellipse function

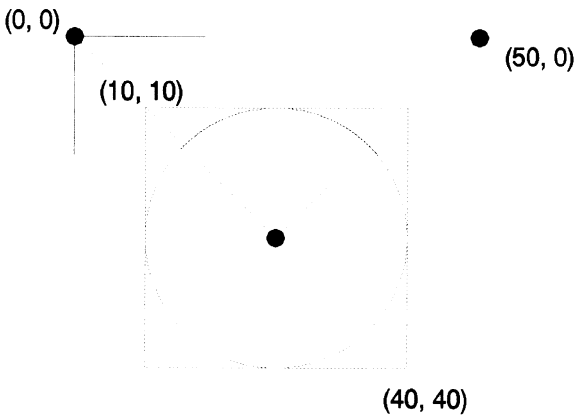


Pie and Chord

The *Pie* and *Chord* functions draw ellipse sections. They both draw an arc just like the *Arc* function. However, *Pie* and *Chord* produce shapes. The *Pie* function connects the ellipse center to the endpoints of the arc. The following *Pie* function draws the top quarter of a circle within the bounding rectangle (10, 10), (40, 40).

```
Pie(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
```

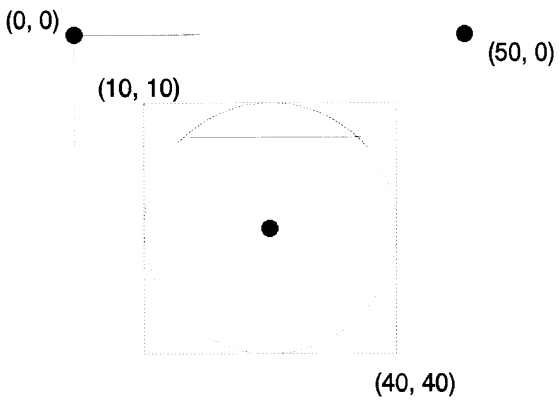
Figure 17.10
The results of the Pie function



The *Chord* function connects the arc's two endpoints together.

```
Chord(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
```

Figure 17.11
The results of the Chord function

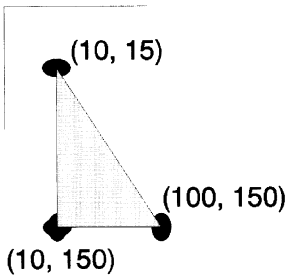


Polygon

The *Polygon* function draws contiguous line segments similarly to the *Polyline* function, except that the *Polygon* function closes the shape by drawing a line segment from the last point specified to the first point specified. Finally, it fills the polygon with the current brush using the current polygon-filling mode. The following *Paint* method paints a right triangle.

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var Points: array[0..2] of TPoint;
begin
  Points[0].X := 10;
  Points[0].Y := 15;
  Points[1].X := 10;
  Points[1].Y := 150;
  Points[2].X := 100;
  Points[2].Y := 150;
  Polygon(PaintDC, @Points, 3);
end;
```

Figure 17.12
The results of the Polygon function



Using palettes

Some types of computer display devices are able to show many colors, but only a few at once. The system or physical palette is the group or selection of colors that are currently available on the device for simultaneous display. Windows gives your application partial control over what colors go into the device's system palette. If your applications use only simple colors, you need not use a palette directly.

However, modifying the system palette affects everything drawn on the screen, including other applications. One application could make all the others appear with the incorrect colors. The Windows palette manager solves this problem by mediating among applications trying to change the system palette. Windows provides each application with a logical palette, which is the group of colors needed by the application. The palette manager maps the requested colors from the logical palette to the available colors in the system palette. If the requested color is not currently listed in the system palette, the palette manager can add the color to the system palette. If the logical palette specifies more colors than the system palette can hold, additional colors are matched to the closest available color in the system palette.

When an application becomes active, it has the option to fill the system palette with colors from its logical palette. This action might force out colors specified by another application's logical palette. In any case, Windows reserves 20 permanent colors in the system palette to generally preserve the color scheme of every application, and Windows itself.

Setting up a palette

Like the pens and brushes described in the "Drawing tools" section, logical palettes are drawing tools. To create a logical palette, use the *CreatePalette* function, which takes a pointer to a *TLogPalette* data record, creates a new palette, and returns a handle to the palette, which can be passed to *SelectPalette* to select the palette into a display context. The *TLogPalette* record contains fields for the Windows version number (currently $0300), the number of palette entries, and an array of palette entries. Each palette entry is a record of type *TPaletteEntry*. The *TPaletteEntry* type has three byte fields for color specification (*peRed*, *peGreen*, and *peBlue*) and one for flags (*peFlags*).

GetStockObject(Default_Palette) creates a default palette containing the 20 colors which are always present in the system palette.

Once the application selects its palette into the display context by using *SelectPalette*, it must "realize" the palette before using it. This is done with the Windows function *RealizePalette*:

```
ThePalette := CreatePalette(@ALogPalette);
SelectPalette(TheDC, ThePalette, 0);
RealizePalette(TheDC);
```

RealizePalette puts the colors from your application's logical palette into the device's system palette. First Windows matches colors which are already in the system palette, then it adds your new colors to the system palette as long as there is room. Finally, colors that couldn't be matched to identical colors in the system palette are matched to the most similar color in the system palette.

Your application should realize its palette before drawing, just as it would select its other drawing tools.

Drawing with palettes

Once your application's palette is realized, your application can draw with its colors. You can specify palette colors either directly or indirectly. To specify a palette color directly, use a palette-index *TColorRef*. A palette-index *TColorRef* is a *Longint* value with the high-order byte set to 1 and the index of a logical-palette entry in the two low-order bytes. For example, $01000009 specifies the ninth entry in a logical palette. This value can be used anywhere a *TColorRef* argument is expected. For example:

```
ALogPen.lopnColor := $01000009;
```

If your display device allows full 24-bit color with no system palette, using a palette index unnecessarily limits you to the colors in your logical palette. To avoid this limitation, you can specify palette colors indirectly by using a palette-relative *TColorRef* value. A palette-relative *TColorRef* is the same as an explicit RGB *TColorRef*, except that the high-order byte is set to 2. The lower three bytes hold the RGB color value. For example, $020000FF would specify a palette-relative *TColorRef* value for pure red. If the device supports a system palette, Windows will match the RGB information to the closest color in the selected logical palette; if the device does not support a system palette, then the *TColorRef* is used as if it specified an explicit RGB value.

Querying a palette

Windows defines a function that allows you to get information about a palette. *GetPaletteEntries* takes an index, a range, and a pointer to a *TPaletteEntry*, and fills the buffer with the specified palette entries.

Modifying a palette

There are two ways to change entries in a logical palette. The *SetPaletteEntries* function takes the same arguments as *GetPaletteEntries* and changes the specified entries to those pointed to by the third argument. Note that changes are not mapped into the system palette until *RealizePalette* is called and don't become visible until the client area is redrawn. The *AnimatePalette* function takes the same arguments as *SetPaletteEntries* but is used when an application wants to change the palette quickly and to make those changes immediately visible. When *AnimatePalette* is called, palette entries with the *peFlags* field set to the constant *pc_Reserved* will be replaced with the corresponding new entries and immediately mapped to the system palette. Other entries will not be affected.

For example, you might want to get the first ten entries in a palette, change their values by increasing their red content and decreasing their blue and green content, and make the changes take effect (assuming some of the palette entries had *pc_Reserved* set):

```
GetObject(ThePalette, SizeOf(NumEntries), @NumEntries);
if NumEntries >= 10 then
begin
  GetPaletteEntries(ThePalette, 0, 10, @PaletteEntries);
  for i := 0 to 9 do
  begin
    PaletteEntries[i].peRed := PaletteEntries[i].peRed + 40;
    PaletteEntries[i].peGreen := PaletteEntries[i].peGreen - 40;
    PaletteEntries[i].peBlue := PaletteEntries[i].peBlue - 40;
  end;
  AnimatePalette(ThePalette, 0, 10, @PaletteEntries);
end;
```

Instead of *AnimatePalette* you could have used:

```
SetPaletteEntries(ThePalette, 0, 10, @PaletteEntries);
RealizePalette(ThePalette);
```

and then redrawn the window to see the colors change.

Responding to palette changes

When a window receives a *wm_PaletteChanged* message, it is an indication that the active window has changed the system palette by realizing its logical palette. The receiving window may respond in one of three ways: It can do nothing (very fast but may lead to incorrect colors), it can realize its logical palette and redraw itself (slower but colors will be as correct as possible), or it can realize its palette and then use the *UpdateColors* function to quickly update the client area to the system palette. *UpdateColors* is generally faster than redrawing the client area, though there may be some loss of color accuracy when it is used. The *WParam* field of the *TMessage* record passed in a *wm_PaletteChanged* message contains the handle of the window that realized its palette. If you realize your logical palette in response, first make sure that this handle is not the handle of your window so you don't create an endless loop.

The program *PalTest* creates and realizes a logical palette with eight colors in it. When you click with the left button it draws a square with each color. When you click with the right button it rotates the colors in the logical palette. A palette-index *TColorRef* is used, so when the logical palette is changed and the boxes redrawn their colors will change. You may find the function *PaletteIndex* useful when using palette-index *TColorRefs*.

The complete file, PALTEST.PAS, can be found on your distribution disks.

C H A P T E R

18

Resources in depth

Windows programs are easy to use because they provide a standard user interface. For example, most Windows programs use menus to let you implement program commands and cursors to let the mouse pointer represent a wide variety of tools, such as arrows or paint brushes.

Menus and cursors are two examples of a Windows program's resources. Resources are data stored in a program's executable (.EXE) file but stored separately from the program's normal data segment. Resources are designed and specified outside the program code, then added to the program's compiled code to create a program's executable file.

These are the resources you will create and use most often:

- Menus
- Dialog boxes
- Icons
- Cursors
- Keyboard accelerators
- Bitmaps
- Character strings

Typically, Windows leaves resources on disk when it loads an application into memory and loads individual resources as it needs them during program execution. Except in the case of bitmaps, when Windows is done with the resource, it discards it from memory. If you want the resource to be loaded when the program is loaded, or if you don't want Windows to be able to

discard the resource from memory, you can change its attributes. See the *Resource Workshop User's Guide* for details on how to create or modify resources.

Creating resources

You can create resources using a resource editor or a resource compiler. Pascal supports both methods, so you can choose whichever approach is more suitable. In most cases it is easier to use a resource editor and create your resources visually. However, it is sometimes convenient to use a resource compiler to compile resource script files that appear in books or magazines.

Regardless of which approach you take, you normally create a resource file (.RES) for each application. This resource file will contain binary information for all of the menus, dialogs, bitmaps, and other resources used by your application.

The binary resource file (.RES) is added to your executable file (.EXE) during compilation by using the **$R** compiler directive as described in this chapter. You must also write code that loads the resources into memory. Each resource must be loaded into memory separately. This gives you flexibility, since your program will only use memory for the resources that are currently required. Loading resources into memory is explained in this chapter.

Adding resources to an executable

Once the resources are stored in binary format in a .RES file, they must be added to the program's executable (.EXE) file. The result is a file that contains the application's compiled code as well as its resources.

There are three ways to add resources to an executable file:

- Use the Resource Workshop to copy resources from a .RES file into the program's already-compiled .EXE file. See the *Resource Workshop User's Guide* for instructions.
- Specify a directive in the source code file. For example, this Pascal program:

```
program SampleProgram;
{$R SAMPPROG.RES}
  ⋮
```

adds the resource file SAMPPROG.RES to the executable file. Each Pascal program can have only one resource file (although that resource file can include other resource files). All the files must be .RES files that store the resources in binary format. The **$R** compiler directive allows you to specify a single .RES file.

Loading resources into an application

Once a resource has been added to the executable file, it must be explicitly loaded by the application before making use of it. The specific way to load a resource depends on the resource type.

Loading menus

A window's menu is one of its creation attributes. In other words, it's a characteristic of the window that must be specified before the window's corresponding element is created (by a *Create* method). Therefore, a menu can be specified in the window type's *Init* constructor or sometime soon after construction. Menu resources are loaded by calling the *LoadMenu* Windows function with the menu ID string when a new window object is constructed. For example,

```
constructor SampleMainWindow.Init(AParent: PWindowsObject;
  ATitle: PChar);
begin
  inheritedInit(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, MakeIntResource(100));
    ⋮
end;
```

The code *MakeIntResource(100)* casts the integer 100 to the type *PChar*, the Windows-compatible string type.

LoadMenu loads the menu resource with an ID of 100 into the new window object. A resource can have a symbolic name (a string), such as 'SampleMenu', rather than a numerical identifier. In that case, the previous code would look like this:

```
constructor SampleMainWindow.Init(AParent: PWindowsObject,
  ATitle: PChar);
```

```
begin
  inheritedInit(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, 'SampleMenu');
   ⋮
end;
```

For more information on creating window objects, see Chapter 10, "Window objects."

To process a menu selection, simply define a method for the window that owns the menu, using the special method definition header extension with the *cm_First* identifier:

```
procedure HandleMenu101(var Msg: TMessage); virtual cm_First + 101;
```

Note that 101 is the ID of the menu item selected (not of the menu resource itself). Each menu item has a unique integer ID. In this method, have the tasks appropriately respond to the menu selection. Usually, you will want to define symbolic constants for your menu commands.

Loading accelerators

Accelerators are hot keys, or keyboard shortcuts, used to issue an application command. Typically, accelerators are defined as substitutes for menu choices. For example, the delete key is a standard accelerator that can be used as an alternative to choosing Delete from an Edit menu. Accelerators can, however, implement commands that don't correspond to menu items.

Accelerator resources are stored in an accelerator table. To load an accelerator table, use the *LoadAccelerators* Windows function, which simply returns a handle to the table. Unlike a menu resource, which is associated with a particular window, an accelerator resource belongs to the entire application. Each application can only have one. Application objects reserve one object field, *HAccTable*, to store a handle to the accelerator resource. Usually, you load the accelerator resource in the application object's *InitInstance* method:

```
procedure TSampleApplication.InitInstance;
begin
  inherited InitInstance;
  HAccTable := LoadAccelerators(HInstance, 'SampleAccelerators');
end;
```

Often, you want to define accelerator keys as shortcuts for menu selections. For example, *Shift+Ins* is usually used as a shortcut for

Paste. Accelerator selections generate command-based messages identical to the ones generated by menu selections. To associate the menu selection's response method with the corresponding accelerator key, make sure the accelerator's value as defined in the resource is identical to the menu item's ID.

Loading dialog boxes

Dialog boxes are the only resource type to have directly-corresponding ObjectWindows object types. *TDialog* and its descendant types, including *TDlgWindow*, define interface objects that use dialog box resources. Each dialog box object is typically associated with one dialog box resource, which specifies its size, location, and assortment of controls, such as buttons and list boxes.

Specify a dialog box object's resource when the dialog box is constructed. As with menu and accelerator resources, a dialog resource may have a symbolic name or an integer ID. For example,

```
ADlg := New(PSampleDialog, Init(@Self, 'AboutBox'));
```

or

```
ADlg := New(PSampleDialog, Init(@Self, MakeIntResource(120)));
```

For more information on creating dialog box objects, see Chapter 11, "Dialog box objects."

Loading cursors and icons

Each window object type has special attributes called *registration attributes*. Among these attributes are the window's cursors and icons. To set these attributes for a window type, you must define a method called *GetWindowClass* (as well as one called *GetClassName*).

For example, assume you've created a cursor for selecting items in a list box. The cursor looks like an index finger and is stored as a cursor resource named 'Finger'. In addition, you've created an icon resource named 'SampleIcon' that looks like a happy face. You would write the *GetWindowClass* method as follows:

```
procedure TSampleWindow.GetWindowClass(var AWndClass: TWndClass);
begin
  inherited GetWindowClass(AWndClass);
```

```
  AWndClass.hCursor := LoadCursor(HInstance, 'Finger');
  AWndClass.hIcon := LoadIcon(HInstance, 'SampleIcon');
end;
```

One difference between cursors and icons, however, is that the cursor is specified for one window, while the icon represents the entire application. Thus, set the icon in the object type for the main window only. One exception to the one-icon-per-application rule is an application that follows the multiple document interface (MDI) standard, for which each MDI child window has its own icon.

In order to use one of Windows' stock cursors or icons, pass 0 in place of *HInstance* and use an *idc_* value, such as *idc_IBeam*, for cursors or an *idi_* value, such as *idi_Hand*, for icons. For example,

```
procedure TSampleWindow.GetWindowClass(var AWndClass: TWndClass);
begin
  inherited GetWindowClass(AWndClass);
  AWndClass.hCursor := LoadCursor(0, idc_IBeam);
  AWndClass.hIcon := LoadIcon(0, idi_Hand);
end;
```

For more information on window registration attributes, see Chapter 10, "Window objects."

Loading string resources

The principal reason for defining an application's strings as resources is to make it easy to customize the application for particular uses or to translate the application into other languages. If the strings are defined within the source code, you have to tamper with the source code to alter or translate the strings. If they're defined as resources, the strings are stored in a string table within the application's executable file. You can then use the string editor to translate strings in the string table without altering or even accessing the source code. Each executable file can have only one string table.

To load a string from the string table into a buffer in your application's data segment, use the *LoadString* function. The syntax for *LoadString* is

```
LoadString(HInstance, StringID, @TextItem, SizeOf(TextItem));
```

- The *StringID* parameter is the string's ID number, such as 601, in the string table. You can substitute a constant for this number.

- The *@TextItem* parameter is a pointer to a character array (*PChar*) that receives the string.
- The *SizeOf(TextItem)* parameter is the maximum number of characters to transfer into *@TextItem*. The maximum size of a string resource is 255 characters, so passing a buffer of 256 characters guarantees getting the entire string.

LoadString returns the number of characters copied to the buffer, or zero if the resource does not exist.

You can use a string resource to display text within a message box. For example, you might want to display an error message. In this example, assume you define the string 'Program unavailable' in a string table and define the constant *ids_NoPrgrm* for that string's ID. To use that string resource in an error box, you could write the following procedure:

```
procedure TTestDialog.RunErrorBox(ErrorNumber: Integer); virtual;
var TextItem: array[0..255] of Char;
begin
  LoadString(HInstance, ids_NoPrgrm, @TextItem, 20);
  MessageBox(HWindow, @TextItem, 'Error', mb_OK or
    mb_IconExclamation);
end;
```

This example loads a single string into an error box. To load a list of strings into a list box, call *LoadString* to obtain each string, then call *AddString* to add it to the list box.

Another use for string resources is for menu items that are added or appended to a menu in your source code. In this case, first obtain the string resource with *LoadString*. Then pass the string as a parameter in calls to the Windows functions *CreateMenu* and *AppendMenu*. For example,

```
constructor TSampleWindow.Init(AParent: PWindowsObject;
  ATitle: PChar);
var TextItem: array[0..255] of Char;
begin
  inherited Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, MakeIntResource(100));
  LoadString(HInstance, 301, @TextItem, 20);
  AppendMenu(Attr.Menu, mf_String or mf_Enabled, 501, @TextItem);
end;
```

Loading bitmaps

The *LoadBitmap* Windows functions load bitmap resources. *LoadBitmap* loads the bitmap into memory and returns its handle. For example,

```
HMyBit := LoadBitmap(HInstance, MakeIntResource(501));
```

loads the bitmap resource identified by 501 and stores the resulting bitmap handle in the variable *HMyBit*. Once a bitmap is loaded, it will stay in memory until you explicitly delete it. Unlike the other resources, it will remain even after the user has closed your application.

Windows supplies a number of predefined bitmaps that are used as part of the Windows graphical interface. Your application can load these bitmaps, such as *obm_DnArrow*, *obm_Close*, and *obm_Zoom*. Like predefined icons and cursors, predefined bitmaps can be loaded by substituting zero for *HInstance* in the *LoadBitmap* call:

```
HMyBit := LoadBitmap(0, MakeIntResource(obm_Close));
```

Once the bitmap is loaded, it can be used in your application in several principal ways:

- To draw a picture on the display. For example, you can load a bitmap into the application's About box to draw the application's logo.
- To create a brush you can use to fill an area of the display or to serve as a window's background. By creating a brush with a bitmap, you can fill a background area with a striped pattern, for example.
- To display pictures rather than text for menu items, or items in a list box. For example, you might display the picture of an arrow for a menu selection, rather than displaying the menu text 'Arrow'.

For more information about using bitmap graphics, see Chapter 17, "The Graphics Device Interface."

You should delete a bitmap from memory when it is not in use. Otherwise, the memory it takes up is unavailable to other applications. If you don't delete it as soon as your application is finished with it, you should delete it before the application terminates.

Delete the bitmap from memory with the Windows function *DeleteObject*:

```
if DeleteObject(HMyBit) then { successful };
```

Once the bitmap is deleted, the handle is invalidated and cannot be used.

Using a bitmap to create a brush

You can use bitmaps to create brushes that can fill an area on the display. The area can be filled with a solid color, or it can form a pattern.

The minimum size of a bitmap used as a brush is 8 by 8 pixels. If you use a larger bitmap, only the upper-left 8 by 8 corner is used in the brush.

Assume you want to fill an area with the striped pattern, as shown in Figure 18.1. To do so, you can use a resource editor to create the pattern of two stripes shown in Figure 18.2, which can be as small as 8 by 8 pixels. Then, in your source code, you can load the bitmap and create a brush with it. To fill the entire area shown in Figure 18.1, Windows copies the brush repeatedly.

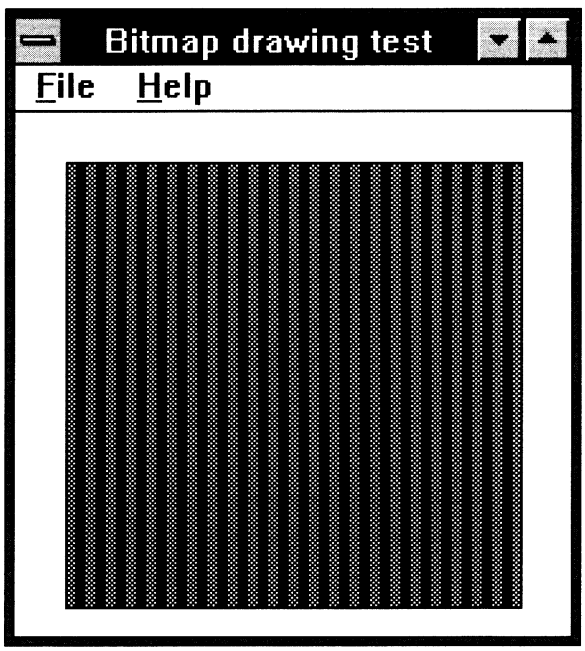


Figure 18.1
Striped pattern filling an area on the display

The actual bitmap is only 8 by 8 pixels, but the brush can be used to fill the entire display.

Figure 18.2
Bitmap resource to create a brush for the pattern in Figure 18.1

The following code sets the bitmap pattern into the brush:

```
procedure TSampleWindow.MakeBrush;
var MyLogBrush: TLogBrush;
begin
  HMyBit := LoadBitmap(HInstance, MakeIntResource(502));
  MyLogBrush.lbStyle := bs_Pattern;
  MyLogBrush.lbHatch := HMyBit;
  TheBrush := CreateBrushIndirect(@MyLogBrush);
end;
```

To test the pattern, display it in a rectangle:

```
procedure MyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
begin
  SelectObject(PaintDC, TheBrush);
  Rectangle(PaintDC, 20, 20, 200, 200);
end;
```

You should delete the bitmap and the brush after using the brush:

```
DeleteObject(HMyBit);
DeleteObject(TheBrush);
```

Displaying bitmaps in menus

To display a bitmap in a menu, use the *ModifyMenu* function. It changes an existing item so it displays a bitmap rather than the menu text defined for the item in a Menu editor. For example, this *Init* constructor adds and modifies a window's menu:

```
type
  MyLong = record
    case Integer of
      0: (TheLong: Longint);
      1: (Lo: Word;
          Hi: Word);
  end;

constructor SampleWindow.Init(AParent: PWindowsObject;
  ATitle: PChar);
var ALong: MyLong;
```

```
begin
  inherited Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, MakeIntResource(100));
  ALong.Lo := LoadBitmap(HInstance, MakeIntResource(503));
  ModifyMenu(Attr.Menu, 111, mf_ByCommand or mf_Bitmap, 211,
    MakeIntResource(ALong.TheLong));
  :
end;
```

In the above code, 111 is the command ID of the menu selection to be changed, and 211 is its new ID. You can use the same ID for both, however.

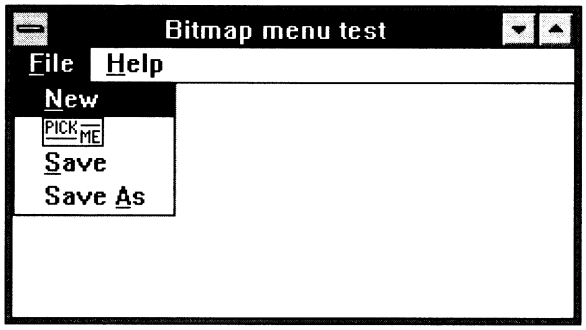


Figure 18.3
A menu that uses a bitmap as one of its selections

C H A P T E R

19

Collections

Pascal programmers traditionally spend much programming time creating code that manipulates and maintains data structures, such as linked lists and dynamically sized arrays. Virtually the same data structure code tends to be written and debugged again and again.

As powerful as traditional Pascal is, it only provides you with built-in record and array types. Any structure beyond that is up to you.

For example, if you're going to store data in an array, you typically need to write code to create the array, to import data into the array, to extract array data for processing, and perhaps to export data to I/O devices. Later, when the program needs a new array element type, you start all over again.

Wouldn't it be great if an array type came with code that would handle many of the operations you normally perform on an array? An array type that could also be extended without disturbing the original code?

That's the aim of the ObjectWindows type, *TCollection*. It's an object that stores a collection of pointers and provides a host of methods for manipulating them.

Collection objects

Besides being an object, and therefore having methods built into it, a collection has two additional features that address the shortcomings of ordinary Pascal arrays—it is dynamically sized and polymorphic.

Collections are dynamically sized

The size of a standard Pascal array is fixed at compile time, which is fine if you know exactly what size your array will always need to be, but it may not be a particularly good fit by the time someone is actually running your code. Changing the size of an array requires changing the code and recompiling.

With a collection, however, you set an initial size, but it can dynamically grow at run time to accommodate the data stored in it. This makes your application much more flexible in its compiled form. Keep in mind, though, that a collection cannot shrink, so you need to be conscious of not making it excessively large.

Collections are polymorphic

A second aspect of arrays that can be limiting to your application is the fact that each element in the array must be of the same type, and that type must be determined when the code is compiled.

Collections get around this limitation by using untyped pointers. Not only is this fast and efficient, but a collection can then consist of objects (and even non-objects) of different types and sizes. A collection doesn't need to know anything about the objects it is handed; it just holds on to them and gives them back when asked.

Type checking and collections

A collection is an end-run around Pascal's traditional strong type checking. That means that you can put anything into a collection, and when you take something back out, the compiler has no way to check your assumptions about what that something is. You can put in a *PHedgehog* and read it back out as a *PSheep*, and the collection will have no way of alerting you.

As a Pascal programmer, you may rightfully feel nervous about such an end-run. Pascal's type checking, after all, saves hours and hours of hunting for some very elusive bugs. So you should

proceed with caution here: You may not even be aware of how difficult a mixed-type bug can be to find, because the compiler has been finding all of them for you! However, if you find that your programs are crashing or locking up, carefully check the types of objects being stored in and read from collections.

Collecting non-objects

You can even add something to a collection that isn't an object at all, but this raises another serious point of caution. Collections expect to receive untyped pointers to something. But some of *TCollection*'s methods act specifically on a collection of *TObject*-derived instances. These include the stream access methods *PutItem* and *GetItem* as well as the standard *FreeItem* procedure.

This means that you can store a *PChar* in a collection, for example, but if you try to send that collection to a stream, the results aren't going to be pretty unless you override the collection's standard *GetItem* and *PutItem* methods. Similarly, when you attempt to deallocate the collection, it will try to dispose of each item using *FreeItem*. If you plan to use non-*TObject* items in a collection, you need to redefine the meaning of "item" in *GetItem*, *PutItem*, and *FreeItem*. That is precisely what *TStrCollection*, for example, does.

If you proceed with prudence, you will find collections (and the descendants of collections that you build) to be fast, flexible, dependable data structures.

Creating a collection

Creating a collection is really just as simple as defining the data type you wish to collect. Suppose you're a consultant, and you want to store and retrieve the account number, name, and phone number of each of your clients. First you define the client object (*TClient*) that will be stored in the collection:

Remember to define a pointer type for each new object type.

```
type
  PClient = ^TClient;
  TClient = object(TObject)
    Account, Name, Phone: PChar;
    constructor Init(NewAccount, NewName, NewPhone: PChar);
    destructor Done; virtual;
    procedure Print; virtual;
  end;
```

Next you implement the *Init* and *Done* methods to allocate and

dispose of the client data and a *Print* method to display the client's data in tabular form. Note that the object fields are of type *PChar* so that memory is only allocated for the portion of the string that is actually used. The *StrNew* and *StrDispose* functions handle dynamic strings very efficiently.

```
constructor TClient.Init(NewAccount, NewName, NewPhone: PChar);
begin
  Account := StrNew(NewAccount);
  Name := StrNew(NewName);
  Phone := StrNew(NewPhone);
end;

destructor TClient.Done;
begin
  StrDispose(Account);
  StrDispose(Name);
  StrDispose(Phone);
end;

procedure TClient.Print;
begin
  Writeln('  ',
    Account, '':10 - StrLen(Account),
    Name, '':20 - StrLen(Name),
    Phone, '':16 - StrLen(Phone));
end;
```

TClient.Done is called automatically for each client when you dispose of the entire collection. Now you just instantiate a collection to store your clients and insert the client records into it. The main body of the program looks like this:

This is COLLECT1.PAS.

```
var
  ClientList: PCollection;

begin
  ClientList := New(PCollection, Init(10, 5));
  with ClientList^ do
  begin
    Insert(New(PClient, Init('91-100', 'Anders, Smitty',
      '(406) 111-2222')));
    Insert(New(PClient, Init('90-167', 'Smith, Zelda',
      '(800) 555-1212')));
    Insert(New(PClient, Init('90-177', 'Smitty, John',
      '(406) 987-4321')));
    Insert(New(PClient, Init('90-160', 'Johnson, Agatha',
     '(302) 139-8913')));
  end;
  PrintAll(ClientList);
```

```
  SearchPhone(ClientList, '(406)');
  Dispose(ClientList, Done);
end.
```

PrintAll and SearchPhone are procedures that are discussed in the next section.

Notice how easy it was to build the collection. The first statement allocates a new *TCollection* called *ClientList* with an initial size of 10 clients. If more than ten clients are inserted into *ClientList*, its size will increase in increments of five clients whenever needed. The next two statements create a new client object and insert it into the collection. The *Dispose* call at the end frees the entire collection—clients and all.

Nowhere did you have to tell the collection what *kind* of data it was collecting—it just took a pointer.

Iterator methods

Insert and deleting items aren't the only common collection operations. You might find yourself writing **for** loops to range over *all* the objects in the collection to display the data or perform some calculation. Other times, you might want to find the first or last item in the collection that satisfies some search criterion. For these purposes, collections have three *iterator* methods: *ForEach*, *FirstThat*, and *LastThat*. Each of these takes a pointer to a procedure or function as its only parameter.

The ForEach iterator

ForEach takes a pointer to a procedure. The procedure has one parameter, which is a pointer to an item stored in the collection. *ForEach* calls that procedure once for each item in the collection, in the order that the items appear in the collection. The *PrintAll* procedure in *Collect1* shows an example of a *ForEach* iterator.

```
procedure PrintAll(C: PCollection);

  procedure CallPrint(P: PClient); far;
  begin
    P^.Print;                                  { Call Print method }
  end;

begin { Print }
  Writeln;
  Writeln;
```

```
  Writeln('Client list:');
  C^.ForEach(@CallPrint);                    { Print each client }
end;
```

For each item in the collection passed as a parameter to *PrintAll*, the nested procedure *CallPrint* is called. *CallPrint* simply prints the client object information in formatted columns.

Iterators must call far local procedures.

You need to be careful about what sort of procedures you call with iterators. To be called by an iterator, a routine (in this example, *CallPrint*) must

- be a procedure—it cannot be a function or an object's method, although as this example showed, the procedure can *call* a method.
- be local to (nested inside) the routine that is calling it.
- be declared as a far procedure, either with the **far** directive or with the {**$F+**} compiler directive.
- take a pointer to a collection item as its only parameter.

The FirstThat and LastThat iterators

In addition to being able to apply a procedure to every element in the collection, it is often useful to be able to find a particular element in the collection based on some criterion. That is the purpose of the *FirstThat* and *LastThat* iterators. As their names imply, they search the collection in opposite directions until they find an item meeting the criteria of the Boolean function passed as an argument.

FirstThat and *LastThat* return a pointer to the first (or last) item that matches the search conditions. Consider the earlier example of the client list and imagine that you can't remember a client's account number or exactly how his last name is spelled. Luckily, you distinctly recall that this was the first client you acquired in the state of Montana. Thus you want to find the first occurrence of a client in the 406 area code (since your list happens to be in chronological order). Here's a procedure using the *FirstThat* method that would do the job:

```
procedure SearchPhone(C: PCollection; PhoneToFind: PChar);

  function PhoneMatch(Client: PClient): Boolean; far;
  begin
    PhoneMatch := StrPos(Client^.Phone, PhoneToFind) <> nil;
  end;
```

```
var
  FoundClient: PClient;

begin { SearchPhone }
  Writeln;
  FoundClient := C^.FirstThat(@PhoneMatch);
  if FoundClient = nil then
    Writeln('No client met the search requirement')
  else
  begin
    Writeln('Found client:');
    FoundClient^.Print;
  end;
end;
```

Again, notice that *PhoneMatch* is nested and uses the **far** call model. In this case, it's a function that returns *True* only if the client's phone number and the search pattern match. If no object in the collection matches the search criteria, a **nil** pointer is returned by *FirstThat*.

Remember: *ForEach* calls a user-defined procedure, while *FirstThat* and *LastThat* each call a user-defined Boolean function. In all cases, the user-defined procedure or function is passed a pointer to an object in the collection.

Sorted collections

Sometimes you need to have your data in a certain order. Object-Windows provides a special type of collection that allows you to order your data in any manner you want: the *TSortedCollection*.

TSortedCollection is a descendant of *TCollection* which automatically sorts the objects it is given. It also automatically checks the collection for duplicate keys when a new member is added.

A *Boolean* field, *Duplicates*, controls whether duplicate keys are allowed. If *Duplicates* is set to *False* (the default), a new member added to the collection replaces an existing member with the same key. When *Duplicates* is *True*, the new member is simply inserted into the collection.

TSortedCollection is an abstract type. To use it, you must first decide what type of data you're going to collect and define two methods to meet your particular sorting requirements. To do this,

you need to derive a new type from *TSortedCollection*. In this case, call it *TClientCollection*.

Your *TClientCollection* already knows how to do all the real work of a collection. It can *Insert* new client records and *Delete* existing ones—it inherited all this basic behavior from *TCollection*. All you have to do is teach *TClientCollection* which field to use as a sort key and how to compare two clients and decide which one belongs ahead of the other in the collection. You do this by overriding the *KeyOf* and *Compare* methods and implementing them as shown here:

```
PClientCollection = ^TClientCollection;
TClientCollection = object(TSortedCollection)
  function KeyOf(Item: Pointer): Pointer; virtual;
  function Compare(Key1, Key2: Pointer): Integer; virtual;
end;

function TClientCollection.KeyOf(Item: Pointer): Pointer;
begin
  KeyOf := PClient(Item)^.Account;
end;

function TClientCollection.Compare(Key1, Key2: Pointer): Integer;
begin
  Compare := StrIComp(PChar(Key1), PChar(Key2));
end;
```

Keys must be typecast because they are untyped pointers.

KeyOf defines which field or fields should be used as a sort key. In this case, it's the client's *Account* field. *Compare* takes two sort keys and determines which one should come first in the sorted order. *Compare* returns –1, 0, or 1, depending on whether *Key1* is less than, equal to, or greater than *Key2*, respectively. This example uses a case-insensitive alphabetical sort of the key (*Account*) strings by calling the *Strings* unit's *StrIComp* function. You could easily sort the collection by names, instead of account numbers, just by changing *KeyOf* to return the *Name* field.

Note that since the keys returned by *KeyOf* and passed to *Compare* are untyped pointers, you need to typecast them into *PChar*s before dereferencing them or passing them to *StrIComp*, in this example.

That's all you have to define! Now if you redefine *ClientList* as a *PClientCollection* instead of a *PCollection* (changing the **var** declaration and the *New* call), you can easily list your clients in alphabetical order:

This completes COLLECT2.PAS.

```
var
  ClientList: PClientCollection;
   ⋮
begin
  ClientList := New(PClientCollection, Init(10, 5));
   ⋮
end.
```

Notice also how easy it would be if you wanted the client list sorted by account number instead of by name. All you would have to do is change the *KeyOf* method to return the *Account* field instead of the *Name* field.

String collections

Many programs need to keeping track of sorted strings. For this purpose, ObjectWindows provides a special-purpose collection, *TStrCollection*. Note that the elements in a *TStrCollection* are *not* objects—they are pointers to null-terminated strings. Since a string collection is a descendant of *TSortedCollection*, duplicate strings can be stored.

There is also a TStringCollection type defined for storing Pascal strings.

Using a string collection is easy. Just declare a pointer variable to hold the string collection. Allocate the collection, giving it an initial size and an amount to grow by as more strings are added:

This is COLLECT3.PAS.

```
var
  WordList: PCollection;
  WordRead: PChar;
   ⋮

begin
  WordList := New(PStrCollection, Init(10, 5));
   ⋮
```

WordList holds ten strings initially and then grows in increments of five. All you have to do is insert some strings into the collection. In this example, words are read out of a text file and inserted into the collection:

```
repeat
   ⋮
  if GetWord(WordRead, WordFile)^ <> #0 then
    WordList^.Insert(StrNew(WordRead));
```

```
  ⋮
until WordRead[0] = #0;
⋮
Dispose(WordList, Done);
```

Notice that the *StrNew* function is used to make a copy of the word that was read and the address of the string copy is passed to the collection. When using a collection, you always give it control over the data you're collecting. It will take care of deallocating the data when you're done. And that's exactly what the call to *Dispose* does: It disposes of each element in the collection, then disposes of the *WordList* collection itself.

Iterators revisited

The *ForEach* method traverses the entire collection one item at a time and passes each one to a procedure you provide. Continuing with the previous example, the procedure *PrintWord* is given a pointer to a string to display. Note that *PrintWord* is a nested (or local) procedure. Wrapped around it is another procedure, *Print*, which is given a pointer to a *TStrCollection*. Print uses the *ForEach* iterator method to pass each item in its collection to the *PrintWord* procedure.

```
procedure Print(C: PCollection);

  procedure PrintWord(P: PChar); far;
  begin
    Writeln(P);                                  { Display the string }
  end;

begin { Print }
  Writeln;
  Writeln;
  C^.ForEach(@PrintWord);                          { Call PrintWord }
end;
```

PrintWord should look familiar. It's just a procedure that takes a string pointer and passes its value to *Writeln*. Note the **far** directive after *PrintWord*'s declaration. *PrintWord* cannot be a method—it must a procedure. And it must be a nested procedure as well. Think of *Print* as a wrapper around a procedure that has the job of doing something—displaying or modifying data, perhaps—with each item in the collection. You can have more than one procedure like the preceding *PrintWord*, but each has to be nested inside *Print* and each has to be a far procedure (using the **far** directive or {**$F+**}).

Finding an item

Sorted collections (and therefore string collections) have a *Search* method that returns the index of an item with a particular key. But how do you find an item in a collection that may not be sorted? Or when the search criteria don't involve the key itself? The answer, of course, is to use *FirstThat* and *LastThat*. You simply define a Boolean function to test for whatever criteria you want, and call *FirstThat*.

Polymorphic collections

You've seen that collections can store any type of data dynamically, and there are plenty of methods to help you access collection data efficiently. In fact, *TCollection* itself defines 23 methods. When you use collections in your programs, you'll be equally impressed by their speed. They're designed to be flexible and implemented to be fast.

But now comes the *real* power of collections: Items can be treated polymorphically. That means you can do more than just store an object type on a collection. You can store many different objects types, from anywhere in your object hierarchy.

If you consider the collection examples you've seen so far, you'll realize that all the items on each collection were of the same type. There was a list of strings in which every item was a string. And there was a collection of clients. But collections can store *any* object that is a descendant of *TObject*, and you can mix these objects freely. Naturally, you want the objects to have something in common. In fact, you want them to have an abstract ancestor object in common.

As an example, here's a program that puts three different graphical objects into a collection. Then a *ForEach* iterator is used to traverse the collection and display each object.

Unlike the other examples in this chapter, this example (*Collect4*) uses Windows functions to draw itself in a window. Be sure to include the *WinProcs* and *WinTypes* units in your **uses** clause for this example.

The abstract ancestor object is defined first:

This is from COLLECT4.PAS.

```
type
  PGraphObject = ^TGraphObject;
```

```
TGraphObject = object(TObject)
  Rect: TRect;
  constructor Init(Bounds: TRect);
  procedure Draw(DC: HDC); virtual;
end;
```

You can see from this declaration that each graphical object can initialize itself (*Init*) and display itself on the graphics screen (*Draw*). Now define an ellipse, a rectangle, and a pie slice, each derived from this common ancestor:

```
PGraphEllipse = ^TGraphEllipse;
TGraphEllipse = object(TGraphObject)
  procedure Draw(DC: HDC); virtual;
end;

PGraphRect = ^TGraphRect;
TGraphRect = object(TGraphObject)
  procedure Draw(DC: HDC); virtual;
end;

PGraphPie = ^TGraphPie;
TGraphPie = object(TGraphObject)
  ArcStart, ArcEnd: TPoint;
  constructor Init(Bounds: TRect);
  procedure Draw(DC: HDC); virtual;
end;
```

These three object types all inherit the *Rect* field from *TGraphObject*, but they are all different sizes. *TGraphEllipse* and *TGraphRect* each need only add a new drawing method, since their drawing methods only need size and position information, while *TGraphPie* needs extra fields and a different constructor to be able to represent itself correctly. Here's the code to put these miscellaneous figures into the collection:

```
⋮
GraphicsList := New(PCollection, Init(10, 5));   { Create collection }

for I := 1 to NumToDraw do
begin
  case I mod 3 of                                  { Create an object }
    0: P := New(PGraphRect, Init(Bounds));
    1: P := New(PGraphEllipse, Init(Bounds));
    2: P := New(PGraphPie, Init(Bounds));
  end;
  GraphicsList^.Insert(P);                    { Add it to collection }
end;
⋮
```

As you can see, the **for** loop inserts graphical objects into the *GraphicsList* collection. All you know is that each object in *GraphicsList* is some kind of *TGraphObject*. But once inserted, you have no idea whether a given item in the collection is a rectangle, ellipse, or pie slice. Thanks to polymorphism, you don't need to know, since each object contains the data and the code (*Draw*) it needs. Just traverse the collection using an iterator method and have each object display itself:

```
procedure DrawAll(C: PCollection);

  procedure CallDraw(P: PGraphObject); far;
  begin
    P^.Draw(PaintDC);                          { Call the Draw method }
  end;

begin { DrawAll }
  C^.ForEach(@CallDraw);                          { Draw each object }
end;

var
  GraphicsList: PCollection;
begin
   ⋮
  if GraphicsList <> nil then DrawAll(GraphicsList);
   ⋮
end.
```

This ability of a collection to store different but related objects leans on one of the powerful cornerstones of object-oriented programming. In the next chapter, you'll see this same principal of polymorphism applied to streams with equal advantage.

Collections and memory management

A *TCollection* can grow dynamically from the initial size set by *Init* to a maximum size of 16,380 elements. The maximum collection size is stored by ObjectWindows in the variable *MaxCollectionSize*. Each element you add to a collection only takes four bytes of memory, because the element is stored as a pointer.

No library of dynamic data structures would be complete unless it provided some provision for error detection. If there is not enough memory to initialize a collection, a **nil** pointer is returned.

If memory is not available when adding an element to a collection, the method *TCollection.Error* is called and a run-time heap

memory error occurs. You may want to override *TCollection.Error* to provide your own error reporting or recovery mechanism.

You need to pay special attention to heap availability, because the user has much more control of an ObjectWindows program than a traditional Pascal program. If the user is the one who controls the adding of objects to a collection (for example, by opening new windows on the desktop), the possibility of a heap error may not be so easy to predict. You may need to take steps to protect the user from a fatal run-time error, with either memory checks of your own when a collection is being used, or a run-time error handler that lets the program recover gracefully.

C H A P T E R

20

Streams

Object-oriented programming techniques and ObjectWindows give you a powerful way of encapsulating code and data, and powerful ways of building an interrelated structure of objects. But what if you want to do something simple, like store some objects on disk?

Back in the days when data sat by itself in a record, writing data to disk was pretty clear-cut, but the data within an Object-Windows program is largely bound up within objects. You could, of course, separate the data from the object and write the data to a disk file. But you've achieved something important by joining the two together in the first place, and it would be a step backwards to take them apart.

Couldn't OOP and ObjectWindows themselves somehow be enlisted in solving this problem? That's what streams are all about.

A stream in ObjectWindows is a collection of objects on its way somewhere: typically to a file, EMS, a serial port, or some other device. Streams handle I/O on the object level rather than the data level. When you extend an ObjectWindows object, you need to provide for handling any additional data fields that you define. All the complexity of handling the object representation is taken care of for you.

The question: Object I/O

As a Pascal programmer, you know that before you can do any file I/O, you must tell the compiler what type of data you will be reading or writing to the file. The file must be typed, and the type must be determined at compile time.

Pascal implements a very useful workaround to this rule: an untyped file accessed with *BlockWrite* and *BlockRead*. But the lack of type checking creates some extra responsibilities for the programmer, although it does let you perform very fast binary I/O.

A second problem, though, is that you can't use files directly with objects. Pascal doesn't allow you to create a typed file of objects. And because objects may contain virtual methods who addresses are determined at run-time, storing the virtual method information outside the program is pointless. Reading such information *into* a program is even more so.

Again, you can work around the problem. You can copy the data out of your objects and store the information in some sort of file, then rebuild the objects from the raw data again later. But that is a rather inelegant solution at best, and complicates the construction of objects.

The answer: Streams

ObjectWindows allows you to overcome both of these difficulties, and gives you some side benefits as well. Streams provide a simple, yet elegant, means of storing object data outside your program.

Streams are polymorphic

An ObjectWindows stream gives you the best of both typed and untyped files: Type checking is still there, but what you intend to send to a stream doesn't have to be determined at compile time. The reason is that streams know they are dealing with objects, so as long as the object is a descendant of *TObject*, the stream can handle it. In fact, different ObjectWindows objects can as easily be written to the same stream as a group of identical objects.

Streams handle objects

All you have to do is define for the stream which objects it needs to handle, so it knows how to match data with VMTs. Then you can put objects onto the stream and get them back effortlessly.

But how can the same stream read and write such widely differing objects as a *TCollection* and a *TDialog* and not even need to know at compile time what objects it is going to be handed? This is *very* different from traditional Pascal I/O. In fact, a stream can even handle new object types that weren't even created when the stream was compiled.

The answer is *registration*. Each ObjectWindows object type (and any new object types you derive from the hierarchy) is assigned a unique registration number. That number gets written to the stream ahead of the object's data. Then, when you go to read the object back from the stream, ObjectWindows gets the registration number first, and, based on that, knows how much data to read and what virtual methods to attach to your data.

Essential stream usage

On a fairly fundamental level, you can think about streams much as you think about Pascal files. At its most basic, a Pascal file can be simply a sequential I/O device. You write things to it, and you read them back. A stream, then, is a *polymorphic* sequential I/O device, meaning that it behaves much like a sequential file, but you can also read or write various types of objects at the current point.

Streams can also (like Pascal files) be viewed as random-access I/O devices, where you seek to a position in the file, read or write at that point, return the position of the file pointer, and so on. These operations are also available with streams, and are described in the section "Random-access streams."

There are two different aspects of stream usage that you need to master, and luckily they are both quite simple. The first is setting up a stream, and the second is reading and writing objects to the stream.

Setting up a stream

All you have to do to use a stream is initialize it. The exact syntax of the *Init* constructor will vary, depending on what type of stream you're dealing with. For example, if you're opening a DOS stream, you need to pass the name of the DOS file and the access mode (read-only, write-only, read/write) for the file containing the stream.

For example, to initialize a buffered DOS stream for loading a collection object into a program, all you need to is this:

```
var
  SaveFile: TBufStream;
begin
  SaveFile.Init('COLLECT.DTA', stOpen, 1024);
   ⋮
```

Once you've initialized the stream, you're ready to go—that's all there is to it.

TStream is an abstract stream mechanism, so you can't actually create an instance of it, but useful stream objects are all derived from *TStream*. These include *TDosStream*, which provides disk I/O, and *TBufStream*, which provides buffered disk I/O (useful if you read or write a lot of small pieces to disk), and *TEmsStream*, a stream that sends objects to EMS memory.

ObjectWindows also implements an indexed stream, with a pointer to a place in the stream. By relocating the pointer, you can do random stream access.

Reading and writing a stream

TStream, the basic stream object, implements three basic methods you need to understand: *Get*, *Put*, and *Error*. *Get* and *Put* roughly correspond to the *Read* and *Write* procedures you would use for ordinary file I/O operations. *Error* is a procedure that gets called whenever a stream error occurs.

Putting it on

Let's look first at *Put*. The general syntax of a *Put* method is this:

```
SomeStream.Put(PSomeObject);
```

where *SomeStream* is any object descended from *TStream* that has been initialized, and *PSomeObject* is a pointer to any object

descended from *TObject* that has been registered with the stream. That's all you have to do. The stream can tell from *PSomeObject*'s VMT what type of object it is (assuming the type has been registered), so it knows what ID number to write, and how much data to write after it.

Of special interest to you as an ObjectWindows programmer, however, is the fact that when you *Put* a parent window with child windows onto a stream, the child windows are automatically written to the stream as well. Thus, saving complex objects is not complex at all—in fact, it's automatic! You can save the entire state of a dialog simply by writing the dialog object onto a stream. When you restart your program and load the dialog back in, it will be in the same condition it was in when you saved it.

Getting it back

Getting objects back from the stream is just as easy. All you have to do is call the stream's *Get* function:

```
PSomeObject := SomeStream.Get;
```

where again, *SomeStream* is an initialized ObjectWindows stream, and *PSomeObject* is a pointer to any type of ObjectWindows object. *Get* simply returns a pointer to whatever it has pulled off the stream. How much data it has pulled, and what type of VMT it has assigned to that data, is determined not by the type of *PSomeObject*, but by the type of object found on the stream. Thus, if the object at the current position of *SomeStream* is not of the same type as *PSomeObject*, you will get garbled information.

As with *Put*, *Get* retrieves complex objects. Thus, if the object you retrieve from a stream is a window that owns child windows, the child windows are loaded as well.

In case of error

Finally, the *Error* procedure determines what happens when a stream error occurs. By default, *TStream.Error* simply sets two fields (*Status* and *ErrorInfo*) in the stream. If you want to do anything fancier, like generating a run-time error or popping up an error dialog box, you need to override the *Error* procedure.

Shutting down the stream

When you're finished using a stream, you call its *Done* method, much as you would normally call *Close* for a disk file. As with any ObjectWindows object, you do this as

```
Dispose(SomeStream, Done);
```

so as to dispose of the stream object as well as shutting it down.

Making objects streamable

All standard ObjectWindows objects are ready to be used with streams, and all ObjectWindows streams know about the standard objects. When you derive a new object type from one of the standard objects, it is very easy to prepare it for stream use and to alert streams to its existence.

Load and Store methods

The actual reading and writing of objects to the stream is handled by methods called *Load* and *Store*. While each object must have these methods to be usable by streams, you never call them directly. (They are called by *Get* and *Put*.) So all you need to do is make sure that your object knows how to send itself to the stream when called upon to do so.

Because of OOP, this job is very easy, since most of the mechanism is inherited from the ancestor object. All your object has to handle is loading or storing the parts of itself that you added. The rest is taken care of by calling the ancestor's method.

For example, let's say you derive a new kind of window from *TWindow*, named after the surrealist painter Rene Magritte, who painted many famous pictures of windows:

```
type
  TMagritte = object(TWindow)
    Surreal: Boolean;
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
    ⋮
  end;
```

All that has been added to the data portion of the window is one Boolean field. In order to load the object, then, you simply read a standard *TWindow*, then read an additional byte to accommodate the Boolean field. The same applies to storing the object. You simply write a *TWindow*, then write one more byte. Typical *Load* and *Store* methods for descendant objects look like this:

```
constructor TMagritte.Load(var S: Stream);
begin
  inherited Load(S);                          { load the ancestor type }
  S.Read(Surreal, SizeOf(Surreal));           { read additional fields }
end;

procedure TMagritte.Store(var S: Stream);
begin
  inherited Store(S);                        { store the ancestor type }
  S.Write(Surreal, SizeOf(Surreal));         { write additional fields }
end;
```

It is entirely your responsibility to ensure that the same amount of data is stored as is loaded, and that data is loaded in the same order that it is stored. The compiler will return no errors. This can cause huge problems if you are not careful. If you modify an object's fields, make sure to update *both* the *Load* and *Store* methods.

Stream registration

In addition to defining the *Load* and *Store* methods for a new object, you also have to register your new object type with the streams. Registration is a simple, two-step process: You define a stream registration record, and you pass it to the procedure *RegisterType*.

ObjectWindows registers all the standard objects, so you only have to register the new objects you define.

To define a stream registration record, just follow the format. Stream registration records are Pascal records of type *TStreamRec*, which is defined as follows:

```
PStreamRec = ^TStreamRec;
TStreamRec = record
  ObjType: Word;
  VmtLink: Word;
  Load: Pointer;
  Store: Pointer;
  Next: Word;
end;
```

By convention, all ObjectWindows stream registration records are given the same name as the corresponding object type, with the initial "T" replaced by an "R." Thus, the registration record for *TCollection* is *RCollection*, and the registration record for *TMagritte* is *RMagritte*. Abstract types such as *TObject* and *TWindowsObject* do not have registration records because there should never be instances of them to store on streams.

Object ID numbers

The *ObjType* field is really the only part of the record you need to think about; the rest is mechanical. Each new type you define needs its own, unique type-identifier number. ObjectWindows reserves the registration numbers 0 through 99 for the standard objects, so your registration numbers can be anything from 100 through 65,535.

It is your responsibility to create and maintain a library of ID numbers for all your new objects that will be used in stream I/O and to make the IDs available to users of your units. As with menu IDs and user-defined messages, the numbers you assign may be completely arbitrary, as long as they are unique and within the specified range.

The automatic fields

The *VmtLink* field is a link to the objects virtual method table (VMT). You simply assign it as the offset of the type of your object:

```
RSomeObject.VmtLink := Ofs(TypeOf(TSomeObject)^);
```

The *Load* and *Store* fields contain the addresses of the *Load* and *Store* methods of your object, respectively.

```
RSomeObject.Load := @TSomeObject.Load;
RSomeObject.Store := @TSomeObject.Store;
```

The final field, *Next*, is assigned by *RegisterType* and requires no intervention on your part. It simply facilitates the internal use of a linked list of stream registration records.

Register here

Once you have constructed the stream registration record, you call *RegisterType* with your record as its parameter. So, to register your new *TMagritte* object for use with streams, you would include the following code:

```
const
  RMagritte: TStreamRec = (
    ObjType: 100;
    VmtLink: Ofs(TypeOf(TMagritte)^);
    Load: @TMagritte.Load;
    Store: @TMagritte.Store
  );

RegisterType(RMagritte);
```

That's all there is to it. Now you can *Put* instances of your new object type to any ObjectWindows stream and *Get* instances back from streams.

Registering standard objects

ObjectWindows defines stream registration records for all its standard objects. In addition, each of the ObjectWindows units defines a registration procedure that automatically registers all of the objects in that unit. For example, *OWindows* has the procedure *RegisterOWindows*, and *ODialogs* has *RegisterODialogs*.

The stream mechanism

Now that you've examined the process you go through to use streams, you should probably take a quick look behind the scenes to see just what ObjectWindows does with your objects when you *Get* or *Put* them. It's an excellent example of objects interacting and using the methods built into each other.

The Put process

When you send an object to a stream with the stream's *Put* method, the stream first takes the VMT pointer from offset 0 of the object and looks through the list of types registered with the streams system for a match. When it finds the match, the stream retrieves the object's registration ID number and writes it to the stream's destination. The stream then calls the object's *Store* method to finish writing the object. The *Store* method makes use of the stream's *Write* procedure, which actually writes the correct number of bytes to the stream's destination.

Your object doesn't have to know anything about the stream—it could be a disk file, a chunk of EMS memory, or any other sort of stream—your object merely says "Write me to the stream," and the stream handles the rest.

The Get process

When you read an object from the stream with the *Get* method, its ID number is retrieved first, and the list of registered types is scanned for a match. When the match is found, the registration record provides the stream with the location of the object's *Load*

method and VMT. The *Load* method is then called to read the proper amount of data from the stream.

Again, you simply tell the stream to *Get* the next object it contains and stick it at the location of the new pointer you specify. Your object doesn't even care what kind of stream it's dealing with. The stream takes care of reading the proper amount of data by using the object's *Load* method, which in turn relies on the stream's *Read* method.

All this is transparent to the programmer, but it shows you how crucial it is to register a type before attempting stream I/O with it.

Handling nil object pointers

You can write a **nil** object to a stream. However, when you do, a word of 0 is written to the stream. On reading an ID word of 0, the stream returns a **nil** pointer. 0 is therefore reserved and cannot be used as a stream object ID number. ObjectWindows reserves stream IDs 0 through 99 for internal use.

Collections on streams: An example

In Chapter 19, "Collections," you saw how a collection could hold different, but related, objects. The same polymorphic ability applies to streams as well, and they can be used to store an entire collection on disk for retrieval at another time or even by another program. Go back and look at COLLECT4.PAS. What more must you do to make that program put the collection on a stream?

The answer is remarkably simple. First, start at the base object, *TGraphObject*, and "teach" it how to store its data (*X* and *Y*) on a stream. That's what the *Store* method is for. Then, similarly define a new *Store* method for any descendant of *TGraphObject* that adds additional fields (*TGraphPie* adds *ArcStart* and *ArcEnd*, for example).

Next, build a registration record for each object type that will actually be stored and register each of those types when your program first begins. And that's it. The rest is just like normal file I/O: declare a stream variable; create a new stream; put the entire collection on the stream with one simple statement; and close the stream.

Adding Store methods

Here are the *Store* methods. Notice that *PGraphEllipse* and *PGraphRect* don't need their own, since they don't add any fields to those they inherit from *PGraphObject*

```
type
  PGraphObject = ^TGraphObject;
  TGraphObject = object(TObject)
    Rect: TRect;
    constructor Init(Bounds: TRect);
    procedure Draw(DC: HDC); virtual;
    procedure Store(var S: TStream); virtual;
  end;

  PGraphEllipse = ^TGraphEllipse;
  TGraphEllipse = object(TGraphObject)
    procedure Draw(DC: HDC); virtual;
  end;

  PGraphRect = ^TGraphRect;
  TGraphRect = object(TGraphObject)
    procedure Draw(DC: HDC); virtual;
  end;

  PGraphPie = ^TGraphPie;
  TGraphPie = object(TGraphObject)
    ArcStart, ArcEnd: TPoint;
    constructor Init(Bounds: TRect);
    procedure Draw(DC: HDC); virtual;
    procedure Store(var S: TStream); virtual;
  end;
```

Implementing the *Store* method is quite straightforward. Each object calls its inherited *Store* method, which stores all the inherited data. Then the stream's *Write* method is called to write the additional data

TGraphObject doesn't call TObject.Store because TObject has no data to store.

```
procedure TGraphObject.Store(var S: TStream);
begin
  S.Write(Rect, SizeOf(Rect));
end;

procedure TGraphPie.Store(var S: TStream);
begin
  TGraphObject.Store(S);
  S.Write(ArcStart, SizeOf(ArcStart));
  S.Write(ArcEnd, SizeOf(ArcEnd));
end;
```

Note that *TStream*'s *Write* method does a binary write. Its first parameter can be a variable of any type, but *TStream.Write* has no way to know how big that variable is. The second parameter provides that information and you should follow the convention of using the standard *SizeOf* function. That way, the compiler can guarantee you're always reading or writing the correct amount of data.

Registration records

Defining a registration record constant for each of the descendent types is our last step. It's a good idea to follow the Object-Windows naming convention of using an R as the initial letter, replacing the type's T.

Remember, each registration record gets a unique object ID number (*ObjType*). ObjectWindows reserves 0 through 99 for its standard objects. It's a good idea to keep track of all your objects stream ID numbers in one central place to avoid duplication.

```
const
  RGraphEllipse: TStreamRec = (
    ObjType: 150;
    VmtLink: Ofs(TypeOf(TGraphEllipse)^);
    Load: nil;                                  { No load method yet }
    Store: @TGraphEllipse.Store);

  RGraphRect: TStreamRec = (
    ObjType: 151;
    VmtLink: Ofs(TypeOf(TGraphRect)^);
    Load: nil;                                  { No load method yet }
    Store: @TGraphRect.Store);

  RGraphPie: TStreamRec = (
    ObjType: 152;
    VmtLink: Ofs(TypeOf(TGraphPie)^);
    Load: nil;                                  { No load method yet }
    Store: @TGraphPie.Store);
```

You don't need a registration record for *TGraphObject* because it's an abstract type and thus won't ever be instantiated or put onto a collection or stream. Each registration record's *Load* pointer is set **nil** here because this example is only concerned with storing data onto a stream. *Load* methods are defined and the registration records are updated in the next example (STREAM2.PAS).

Registering

You must always remember to register each of these records before performing any stream I/O. The easiest way to do this is to wrap them all in one procedure and call it at the very beginning of your program (or in your application's *Init* method)

```
procedure StreamRegistration;
begin
  RegisterType(RCollection);
  RegisterType(RGraphEllipse);
  RegisterType(RGraphRect);
  RegisterType(RGraphPie);
end;
```

Notice that you have to register the *TCollection* (using its *RCollection* record—now you see why naming conventions make programming easier) even though you didn't define *TCollection*. The rule is simple and unforgiving: It's *your* responsibility to register every object type that your program puts onto a stream.

Writing to the stream

All that's left to follow is the normal file I/O sequence of: create a stream; put the data (a collection) onto it; close the stream. You don't have to write a *ForEach* iterator to stream each collection item. You just tell the stream to *Put* the collection on the stream:

This is STREAM1.PAS.

```
var
  :
  GraphicsStream: TBufStream;
begin
  :
  StreamRegistration;                { Register all streamed objects }
  GraphicsStream.Init('GRAPH.STM', stCreate, 1024);
  GraphicsStream.Put(GraphicsList);              { Output collection }
  if GraphicsStream.Status <> 0 then
    Status := em_Stream;
  GraphicsStream.Done;                            { Shut down stream }
end;
```

This creates a disk file that contains all the information needed to "read" the collection back into memory. When the stream is opened and the collection is retrieved (see STREAM2.PAS), all the hidden links between the collection and its items, and objects and their virtual method tables are magically restored. The next section explains how to stream objects that contain links to other objects.

Who gets to store things?

An important caution about streams: The owner of an object is the only one that should write that object to a stream. This caution is similar to one with which you have probably become familiar while using traditional Pascal, that the owner of a pointer is the one that should dispose of the pointer.

In the midst of the complexity of a real-life application, numerous objects will often have a pointer to a particular structure. When the time arrives for stream I/O, you need to decide who "owns" the structure; that owner alone should be the one to send that structure to the stream. Otherwise, you end up with multiple copies in the stream of what was initially just one structure. When you then read the stream, multiple instances of the structure are created, with each of the original objects now pointing at their own personal copy of the structure instead of at the original single structure.

Fields in streams

Many times you'll find it convenient to store pointers to a parent window's child windows in local instance variables (the object's data fields). For example, a dialog box might store pointers to its control objects in mnemonically named fields for easy access (fields like *OKButton* or *FileInputLine*). When that child window is created, the parent window has *two* pointers to the child window, one in the field, and one in the child-window list. If you don't allow for this, reading back the object from a stream results in duplicate instances.

The solution is provided in the *TWindowsObject* methods called *GetChildPtr* and *PutChildPtr*. When storing a field that is also a child window, rather than writing the pointer as if it were just another variable, you call *PutChildPtr*, which stores a reference to the ordinal position of the child window in the parent window's child-window list. This way, when you *Load* the parent window back from the stream, you can call *GetChildPtr*, which makes sure the field and the child-window list point to the same object.

Here's a quick example using *GetChildPtr* and *PutChildPtr* in a simple window:

```
type
  TDemoWindow = object(TWindow)
    Msg: PStatic;
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
  end;

constructor TDemoWindow.Load(var S: TStream);
begin
  TWindow.Load(S);
  GetChildPtr(S, Msg);
end;

procedure TDemoWindow.Store(var S: TStream);
begin
  TWindow.Store(S);
  PutChildPtr(S, Msg);
end;
```

Let's take a look at how this *Store* method differs from a normal *Store*. After storing the window normally, all you have to do is store a reference to the *Msg* field, rather than storing the field itself as you would normally do. The actual button object is stored as a child window of the window when you call *TWindow.Store*. All you have to do in addition is put information on the stream indicating that *Msg* is to point to that child window. The *Load* method does the same thing in reverse, first loading the window and its button child window, then restoring the pointer to that child window to *Msg*.

Sibling window instances

A similar situation can arise when a window has a field that points to one of its siblings. A window is called a sibling window of another if both are owned by the same parent window. For example, imagine that you have an edit control and two radio buttons that control whether the edit control can be activated or not. When the state of a radio button changes, it activates or de-activates the edit control accordingly. Because the *TActivateRadioButton* has to know about an edit control which is also a member of the same window, an edit control is added as an instance variable.

As with child windows, you can run into problems when reading and writing sibling references to streams. The solution, however, is similar. The *TWindowsObject* methods *PutSiblingPtr* and *GetSiblingPtr* provide the means for accessing siblings.

```
type
  TActivateRadioButton = object(TRadioButton)
    EditControl: PEdit;
    :
    constructor Load(var S: TStream);
    procedure Store(var S: TStream); virtual;
    :
  end;

constructor TActivateRadioButton.Load(var S: TStream);
begin
  TRadioButton.Load(S);
  GetPeerPtr(S, EditControl);
end;

procedure TActivateRadioButton.Store(var S: TStream); virtual;
begin
  TRadioButton.Load(S);
  PutPeerPtr(S, EditControl);
end;
```

The only thing to worry about is loading references to sibling windows that have not yet been loaded (that is, they come later in the child-window list, and therefore later on the stream). ObjectWindows handles this automatically, keeping track of all such forward references and resolving them when all the child windows have been loaded. The part you may need to consider is that sibling references are not valid until the entire *Load* has been completed. Because of this, you should not put any code into *Load* methods that makes use of child windows that depend on their sibling windows, as the results will be unpredictable.

Copying a stream

TStream has a method *CopyFrom(S,Count)*, which copies *Count* bytes from the given stream *S*. *CopyFrom* can be used to copy the entire contents of a stream to another stream. If you repeatedly access a disk-based stream, for example, you may want to copy it to an EMS stream for more rapid access:

```
NewStream := New(TEmsStream, Init(OldStream^.GetSize));
OldStream^.Seek(0);
NewStream^.CopyFrom(OldStream, OldStream^.GetSize);
```

Random-access streams

So far, we have dealt with streams as sequential devices: You *Put* objects at the end of a stream, and *Get* them back in the same order. But ObjectWindows provides more capabilities than that. Specifically, it allows you to treat a stream as a virtual, random-access device. In addition to *Get* and *Put*, which correspond to *Read* and *Write* on a file, streams provide features analogous to a file's *Seek*, *FilePos*, *FileSize*, and *Truncate*.

- The *Seek* procedure of a stream moves the current stream pointer to a specified position (in bytes from the beginning of the stream), just like the standard Pascal *Seek* procedure.
- The *GetPos* function is the inverse of the *Seek* procedure. It returns a *Longint* with the current position of the stream.
- The *GetSize* function returns the size of the stream in bytes.
- The *Truncate* procedure deletes all data after the current stream position, making the current position the end of the stream.

While these routines are useful, random access streams require you to keep an index, noting the starting position of each object in the stream. In this case, you might use a collection to hold the index.

Non-objects on streams

You can write things that are not objects onto streams, but you have to use a somewhat different approach to do it. The standard stream *Get* and *Put* methods require that you load or store an object derived from *TObject*. If you want to create a stream of non-objects, go directly to the lower-level *Read* and *Write* procedures, each of which reads or writes a specified number of bytes onto the stream. This is the same mechanism used by *Get* and *Put* to read and write the data for objects. You're simply bypassing the VMT mechanism provided by *Get* and *Put*.

Designing your own streams

This section summarizes the methods and error-handling capabilities of ObjectWindows streams so that you know what you can use to create new types of streams.

TStream itself is an abstract object that must be extended to create a useful stream type. Most of *TStream*'s methods are abstract and must be implemented in your descendant, and some depend upon *TStream* abstract methods. Basically, only the *Error*, *Get*, and *Put* methods of *TStream* are fully implemented. *GetPos*, *GetSize*, *Read*, *Seek*, *SetPos*, *Truncate*, and *Write* must be overridden. If the descendant object type has a buffer, the *Flush* method should be overridden as well.

Stream error handling

TStream has a method called *Error(Code, Info)*, which is called whenever the stream encounters an error. *Error* simply sets the stream's *Status* field to one of the constants listed in Chapter 21, "ObjectWindows reference" under "*stXXXX* constants."

The *ErrorInfo* field is undefined except when *Status* is *stGetError* or *stPutError*. If *Status* is *stGetError*, the *ErrorInfo* field contains the stream ID number of the unregistered type. If *Status* is *stPutError*, the *ErrorInfo* field contains the VMT offset of the type you tried to put onto the stream. You can override *TStream.Error* to generate any level of error handling, including run-time errors.

PART 4

ObjectWindows reference

C H A P T E R

21

ObjectWindows reference

This chapter contains an alphabetical listing of all the standard Object-Windows object types, with explanations of their general purposes, usage, fields, and methods. It also describes all the elements of ObjectWindows that are *not* part of the ObjectWindows standard object hierarchy.

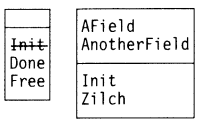
To find information on a specific object, keep in mind that many of the properties of the objects in the hierarchy are inherited from ancestor objects. Rather than duplicate all that information endlessly, this chapter only documents fields and methods that are *new* or *changed* for a particular object. By looking at the inheritance diagram for the object, you can easily determine which of its ancestors introduced a field, and which objects introduce or redefine methods.

The non-object elements listed in this chapter include types, constants, variables, procedures, and functions defined in the ObjectWindows units.

The following are sample reference entries for an object and a procedure:

TSample — TSample's unit

TObject TSample



First is a general overview of the object, it's relationship with other objects, and general usage. The diagram above shows that *TSample* is a direct descendant of *TObject*, and that it overrides the *Init* constructor.

Fields

This section lists all fields for each object alphabetically. In addition to showing the declaration of the field and an explanation of its use, there is a "Read only" or "Read/write" designation. Read-only fields are generally fields that are set up and maintained by the object's methods, and they should *not* be on the left side of an assignment statement.

AField

```
AField: SomeType;
```

Read only

AField is a field that holds some information about this sample object. This text explains how it functions, what it means, and how you use it.

See also: related fields, methods, objects, global functions, etc.

AnotherField

```
AnotherField: Word;
```

Read/write

AnotherField has similar information to that for *AField*.

Methods

This section lists all methods which are either newly defined for this object or which override inherited methods. Constructors are listed first, then destructors, then all other methods in alphabetical order.

For virtual methods, an indication will be given as to how often you will probably need to override the method: Never, Seldom, Sometimes, Often, or Always.

Init

```
constructor Init(AParameter: SomeType);
```

Init creates a new sample object by first calling the *Init* constructor inherited from *TObject*, then setting the *AField* field to *AParameter*.

See also: *TObject.Init*

Zilch

```
procedure Zilch; virtual;
```

Override: Sometimes

The *Zilch* procedure causes the sample object to perform some action.

See also: *TSomethingElse.Zilch*

Sample procedure — Sample's unit

Declaration `procedure Sample(AParameter);`

Function *Sample* performs some useful function on its parameter, *AParameter*.

See also *Example* function

Abstract procedure — Objects

Declaration `procedure Abstract;`

Function A call to this procedure terminates the program with run-time error 211. When implementing an *abstract object type*, use calls to *Abstract* in those virtual methods that must be overridden in descendant types. This ensures that any attempt to use instances of the abstract object type will fail.

AllocMultiSel function — ODialogs

Declaration `function AllocMultiSel(Count: Integer): PMultiSelRec;`

Function Allocates a *TMultiSelRec* with the count equal to *Count*, and enough room in the *Selections* field to hold *Count* selections (0..*Count*-1). Returns **nil** if there is not enough memory to allocate the entire record.

See also *FreeMultiSel*, *TMultiSelRec*

Application variable — OWindows

Declaration `Application: PApplication = nil;`

Function The *Application* variable is set to *@Self* at the beginning of *TApplication.Init* and cleared to **nil** by *TApplication.Done*. Thus, throughout the execution of an ObjectWindows program, *Application* points to the application object.

See also *TApplication.Init*

bf_XXXX constants — ODialogs

Function Button, check box, and radio button objects use *bf_* constants to define their three possible states.

Values The following button flag constants are defined:

Table 21.1 Button flag constants

Constant	Value	Meaning
bf_Unchecked	0	Item is unchecked
bf_Checked	1	Item is checked
bf_Grayed	2	Item is grayed

bs_XXXX Button styles — WinTypes

Function You can pass these constants in the style parameter of button object constructors, or to specify button styles when creating buttons with the *CreateWindow* and *CreateWindowEx* functions.

Values Windows defines the following constants:

Table 21.2 Button styles

Constant	Meaning
bs_3State	The button is a box that toggles among three states: checked, unchecked, and grayed.
bs_Auto3State	The same as *bs_3State*, except the box toggles state automatically when clicked.
bs_AutoCheckBox	The same as *bs_CheckBox*, except the box toggles state automatically when clicked.
bs_AutoRadioButton	The same as *bs_RadioButton*, except that when clicked, the button becomes checked and all other buttons in the same group become unchecked.
bs_CheckBox	The button is a box the user can check and uncheck. Associated text appears to the right of the box.
bs_DefPushButton	The same as *bs_PushButton*, except this button is the default selection unless another button or box is selected using the keyboard or mouse.
bs_GroupBox	The button is a box for grouping other buttons. Associated text appears in the upper left corner.
bs_LeftText	Used with *bs_3State*, *bs_CheckBox*, or *bs_RadioButton*, this style causes associated text to appear to the left of the button or box instead of to the right.
bs_OwnerDraw	The button is an owner-draw button. In addition to normal notification codes sent via *wm_Command* messages, the parent also receives requests to paint, invert, and disable the button.
bs_PushButton	The button is a push button with any associated text placed inside.

Table 21.2: Button styles (continued)

bs_RadioButton	The button is a small round button that the user can check and uncheck. Associated text appears to the right of the button. Radio buttons usually occur in groups with one and only one button checked at a time.

See also *TButton.Init, TCheckBox.Init, TRadioButton.Init*

BWCCClassNames variable — OWindows

Declaration

```
BWCCClassNames: Boolean = False;
```

Function Indicates whether the application uses Borland Windows Custom Controls (BWCC) window-class names for controls. The initialization code of the *BWCC* unit sets *BWCCClassNames* to *True*, so including *BWCC* in the **uses** class of a program or unit automatically sets *BWCCClassNames*. Push button, check box, and radio button objects in ObjectWindows applications that use BWCC automatically use the BWCC class names.

If you want your programs to work with both BWCC and non-BWCC controls, you can load different resources based on the value of *BWCCClassNames*.

See also *GetClassName* methods

cbs_XXXX Combo box styles — WinTypes

Function You can pass these constants in the style parameters of combo box object constructors, or to specify combo box styles when creating combo boxes with the *CreateWindow* and *CreateWindowEx* functions.

Values Windows defines the following values:

Table 21.3
Combo box styles

Constant	Meaning
cbs_AutoHScroll	The text in the edit control scrolls to the left when the user types a character at the end of the line. Without this style, the user cannot type beyond the boundary of the edit control.
cbs_DropDown	The same as *cbs_Simple*, except that the list box appears only when the user clicks the icon next to the selection field.
cbs_DropDownList	The same as *cbs_DropDown*, except a static text item displays the current selection instead of an edit control.

Table 21.3: Combo box styles (continued)

cbs_HasStrings	The combo box uses strings as its entries. Windows manages the strings, which you can retrieve using the *cb_GetLBText* message. Use this style along with *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable*.
cbs_NoIntegralHeight	The combo box appears with exactly the size given at creation. Normally the combo box adjusts it size so it doesn't show partial entries.
cbs_OEMConvert	The combo box translates each character typed in the edit control from the ANSI character set to the OEM character set and then back to ANSI. The *AnsiToOem* API function will then behave correctly when applied to entries in the list box or to the text in the edit control. *cbs_OEMConvert* is useful for combo boxes that contain file names, and can be used with *cbs_Simple* or *cbs_DropDown*.
cbs_OwnerDrawFixed	The combo box's parent window draws the control, with all entries in the list box the same height.
cbs_OwnerDrawVariable	The combo box's parent window draws the control, with entries in the list box of varying heights.
cbs_Simple	The combo box displays its list box at all times. The current list box selection appears in the edit control.
cbs_Sort	The combo box sorts the items in its list box. The sort order can differ for combo boxes with the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* styles.

See also *TComboBox.Init*

cm_XXXX constants — OWindows

Function ObjectWindows defines several constants defining ranges of command message constants.

Values The following command constants are defined:

Table 21.4 Command message constants

Constant	Value	Meaning
cm_First	$A000	Beginning of command messages
cm_Count	$6000	Number of command messages
cm_Internal	$FF00	Beginning of command messages reserved for internal use
cm_Reserved	*cm_Internal – cm_First*	

*cm_* constants are defined for three standard menus: File, Edit, and Window:

Table 21.5 Command offset based default values

Constant	Value	Menu equivalent
cm_EditCut	*cm_Reserved* + 0	Edit | Cut
cm_EditCopy	*cm_Reserved* + 1	Edit | Copy
cm_EditPaste	*cm_Reserved* + 2	Edit | Paste
cm_EditDelete	*cm_Reserved* + 3	Edit | Delete
cm_EditClear	*cm_Reserved* + 4	Edit | Clear
cm_EditUndo	*cm_Reserved* + 5	Edit | Undo
cm_EditFind	*cm_Reserved* + 6	Edit | Search
cm_EditReplace	*cm_Reserved* + 7	Edit | Replace
cm_EditFindNext	*cm_Reserved* + 8	Edit | Search Again
cm_FileNew	*cm_Reserved* + 9	File | New
cm_FileOpen	*cm_Reserved* + 10	File | Open
cm_MDIFileNew	*cm_Reserved* + 11	File | New
cm_MDIFileOpen	*cm_Reserved* + 12	File | Open
cm_FileSave	*cm_Reserved* + 13	File | Save
cm_FileSaveAs	*cm_Reserved* + 14	File | Save As
cm_ArrangeIcons	*cm_Reserved* + 15	Window | Arrange Icons
cm_TileChildren	*cm_Reserved* + 16	Window | Tile
cm_CascadeChildren	*cm_Reserved* + 17	Window | Cascade
cm_CloseChildren	*cm_Reserved* + 18	Window | Close All
cm_CreateChild	*cm_Reserved* + 19	
cm_Exit	*cm_Reserved* + 20	File | Exit

coXXXX constants — Objects

Function The *coXXXX* constants are passed as the *Code* parameter to the *TCollection.Error* method when a *TCollection* detects an error during an operation.

Values The following standard error codes are defined for all ObjectWindows collections:

Table 21.6 Collection error codes

Error code	Value	Meaning
coIndexError	–1	Index out of range. The *Info* parameter passed to the *Error* method contains the invalid index.
coOverflow	–2	Collection overflow. *TCollection.SetLimit* failed to expand the collection to the requested size. The *Info* parameter passed to the *Error* method contains the requested size.

See also *TCollection* object

cs_XXXX Class styles — WinTypes

Function These window class style constants are used in the style field of the *TWndClass* record. You can combine them using **or**.

Values Windows defines the following constants:

Table 21.7 Class styles

Constant	Meaning
cs_ByteAlignClient	The window's client area is aligned on the byte boundary, in the x direction.
cs_ByteAlignWindow	The window is aligned on the byte boundary, in the x direction.
cs_ClassDC	Instances of the window class share their own display context.
cs_DblClks	The window gets mouse double-click messages.
cs_GlobalClass	All running applications can use the window class.
cs_HRedraw	Redraw the entire window if its horizontal size changes.
cs_NoClose	The window's Control menu's Close option is disabled.
cs_OwnDC	Each window instance gets its own display context, using approximately 800 bytes of memory per window.
cs_ParentDC	The window uses its parent's display context.
cs_SaveBits	The window saves its contents to a bitmap when not currently displayed, then uses the bitmap to redisplay the contents. Use minimally.
cs_VRedraw	Redraw the entire window if its vertical size changes.

See also *TWindowsObject.GetWindowClass*

cw_UseDefault constant — WinTypes

cw_UseDefault tells Windows to assign a default size or position to the window being created. You can use *cw_UseDefault* in the *X*, *Y*, *W*, and *H* fields of a window object's *Attr* field, or as a parameter to *CreateWindow* or *CreateWindowEx*.

See also *TWindowAttr* type

DoneMemory procedure — OMemory

Declaration `procedure DoneMemory;`

Function Disposes of the memory allocated to the safety pool.

See also *InitMemory* procedure

em_XXXX constants — OWindows

Function Several standard error conditions are flagged by ObjectWindows constants starting with *em_*.

Values The following error flags are defined:

Table 21.8 Error condition constants

Constant	Value	Meaning
em_InvalidWindow	–1	The window is invalid because *Create* did not succeed.
em_OutOfMemory	–2	A memory allocation ate into the safety pool.
em_InvalidClient	–3	The MDI client window could not be created.
em_InvalidChild	–4	One or more of the window's children is not valid.
em_InvalidMainWindow	–5	The main window could not be created.

EmsCurHandle variable — Objects

Declaration `EmsCurHandle: Word = $FFFF;`

Function Holds the current EMS handle as mapped into EMS physical page 0 by a *TEmsStream*. *TEmsStream* avoids costly EMS remapping calls by caching the state of EMS. If your program uses EMS for other purposes, be sure to set *EmsCurHandle* and *EmsCurPage* to $FFFF before using a *TEmsStream*—this will force the *TEmsStream* to restore its mapping.

See also *TEmsStream.Handle*

EmsCurPage variable — Objects

Declaration

```
EmsCurPage: Word = $FFFF;
```

Function Holds the current EMS logical page number as mapped into EMS physical page 0 by a *TEmsStream*. *TEmsStream* avoids costly EMS remapping calls by caching the state of EMS. If your program uses EMS for other purposes, be sure to set *EmsCurHandle* and *EmsCurPage* to $FFFF before using a *TEmsStream*—this will force the *TEmsStream* to restore its mapping.

See also *TEmsStream.Page*

es_XXXX Edit control styles — WinTypes

Function You can pass these constants in the style parameter of edit control object constructors, or to specify edit control styles when creating edit controls with *CreateWindow* or *CreateWindowEx*.

Values Windows defines the following styles:

Table 21.9
Edit control styles

Constant	Meaning
es_AutoHScroll	The edit control automatically scrolls its text to the left by 10 characters when the user types a character at the end of the line. The text scrolls back to position zero when the user presses *Enter*.
es_AutoVScroll	The edit control automatically scrolls its text up one page when the user presses *Enter* with the caret on the last visible line.
es_Center	The edit control centers its text. Use only with *es_MultiLine*.
es_Left	The edit control aligns its text flush left. Use only with *es_MultiLine*.
es_LowerCase	The edit control converts all characters to lowercase as the user types them.
es_MultiLine	The edit control is a multiple-line edit control. If *es_AutoVScroll* is not used with *es_MultiLine*, a beep sounds when the user presses *Enter* with the caret on the last line. If *es_AutoHScroll* is not used, new words typed in automatically wrap to the next line when necessary. Word wrap positions change when the window changes size. A multiple-line edit control with scroll bars handles its own scroll bar messages, otherwise scrolling is done automatically as described.
es_NoHideSel	The edit control does not hide its selection when it loses the input focus. By default, edit controls hide the selection when they lose the focus.

Table 21.9: Edit control styles (continued)

es_OEMConvert	The edit control converts entered text from the ANSI character set to the OEM character set and then back to ANSI. *AnsiToOem* function will then behave correctly when applied to the edit control's text. *es_OEMConvert* is useful for edit controls that contain file names.
es_Password	All characters typed into the edit control appear as asterisks (*). You can change the displayed character with the *em_SetPasswordChar* message.
es_Right	The edit control aligns its text flush right. Use only with *es_MultiLine*.
es_UpperCase	The edit control converts all characters to uppercase as the user types them.

See also *TEdit.Init*

FreeMultiSel procedure — ODialogs

Declaration

```
procedure FreeMultiSel(P: PMultiSelRec);
```

Frees a *TMultiSelRec* record allocated by *AllocMultiSel*.

See also *AllocMultiSel, TMultiSelRec*

fsFileSpec constant — OStdDlgs

Declaration

```
fsFileSpec = fsFileName + fsExtension;
```

Function Specifies the length of a file name. *TFileDialog* uses *fsFileName* to declare the length of a buffer to hold the name of the file.

id_XXXX constants — OWindows

Function ObjectWindows defines several constants defining ranges of child ID messages.

Values The following child ID message constants are defined:

Table 21.10 Child ID message constants

Constant	Value	Meaning
id_First	$8000	Start of child ID messages
id_Count	$1000	Number of child ID messages
id_Internal	$8F00	Reserved for internal use
id_Reserved	*id_Internal-id_First*	

Table 21.10: Child ID message constants (continued)

id_FirstMDIChild	*id_Reserved* + 1	Base for child-ID numbers
id_MDIClient	*id_Reserved* + 2	Child-ID number of the MDI client window

Function *ExecDialog* and *MessageBox* return the following values to indicate what button the user pressed to close the dialog box or message box. The values are the standard control IDs of common dialog box push button controls.

Values Windows defines the following IDs:

Table 21.11 Dialog box command ID constants

Constant	Meaning
id_Abort	Abort button was pressed.
id_Cancel	Cancel button was pressed.
id_Ignore	Ignore button was pressed.
id_No	No button was pressed.
id_Ok	OK button was pressed.
id_Retry	Retry button was pressed.
id_Yes	Yes button was pressed.

See also *TAppliction.ExecDialog*

InitMemory procedure — OMemory

Declaration

```
procedure InitMemory;
```

Function Initializes the safety pool by calling *RestoreMemory*, then installs a heap error function. *TApplication.Init* calls *InitMemory*.

See also *DoneMemory* procedure, *RestoreMemory* procedure

lbs_XXXX List box styles — WinTypes

Function You can pass these constants in the style parameter of a list box object's constructor, or to specify list box styles when creating list boxes with *CreateWindow* or *CreateWindowEx*.

Values Windows defines the following styles:

Table 21.12
List box styles

Constant	Meaning
lbs_ExtendedSel	The list box allows multiple item selection with *Shift* and the mouse or some other key combination.
lbs_HasStrings	The list box uses strings as its entries. Use with *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable*. Windows manages the strings, which you can retrieve using the *lb_GetText* message.
lbs_MultiColumn	The list box has multiple columns which the user can be scroll horizontally. Set the column width with the *lb_SetColumnWidth* message.
lbs_MultipleSel	The list box allows selection of multiple items with the mouse. Clicking or double-clicking and entry toggles its selection state.
lbs_NoIntegralHeight	The list box appears with the exact size given at creation. Normally the list box resizes itself to not show partial entries.
lbs_NoRedraw	The list box doesn't redraw when changes are made. The *wm_SetRedraw* message sets or resets this style dynamically.
lbs_Notify	The list box sends a message to its parent window whenever the user clicks or double-clicks an entry.
lbs_OwnerDrawFixed	The parent window draws the list box's contents. All items in the list box have the same height.
lbs_OwnerDrawVariable	The parent window draws the list box's contents. Items in the list box vary in height.
lbs_Sort	The list box sorts its entries alphabetically. The sort order may differ for list boxes with the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* styles.
lbs_Standard	The same as the *lbs_Notify* and *lbs_Sort* styles together. The list box has borders on all sides.
lbs_UseTabStops	The list box expands *Tab* characters in its entries. By default there are tab stops each 32 dialog units from the left edge of the entry. A dialog unit is one-fourth of the dialog base width unit, as returned by the *GetDialogBaseUnits* function.
lbs_WantKeyboardInput	The list box sends *wm_VKeyToItem* and *wm_CharToItem* messages sent to its owner when the list box has the input focus and the user presses a key.

See also *TListBox.Init*

LongDiv function — OWindows

Declaration

```
function LongDiv(X: Longint; Y: Integer): Integer;
  inline($59/$58/$5A/$F7/$F9);
```

Function A fast, inline assembly-coded division routine, returning the integer value *X/Y*.

LongMul function — OWindows

Declaration

```
function LongMul(X, Y: Integer): Longint;
  inline($5A/$58/$F7/$EA);
```

Function A fast, inline assembly-coded multiplication routine, returning the long integer value *X* * *Y*.

LongRec type — Objects

Declaration

```
LongRec = record
  Lo, Hi: Word;
end;
```

Function A useful record type for handling double-word length variables.

LowMemory function — OMemory

Declaration

```
function LowMemory: Boolean;
```

Function The *LowMemory* function returns *True* if a memory allocation has eaten into the safety pool at the end of the heap. The size of the safety pool is determined by the *SafetyPoolSize* variable. *LowMemory* is checked automatically by *TApplication.MakeWindow* and *TApplication.ExecDialog*, which should be used for creating window elements. Major consumers of memory (such as large, complex dialog boxes) should check *LowMemory* for themselves periodically to ensure that they don't exceed the available space.

See also *MemAlloc*, *SafetyPoolSize*, *TApplication.ValidWindow*

MakeIntResource type — WinTypes

Declaration

```
MakeIntResource = PStr;
```

Function In order to specify a resource by its ID number, you must typecast the number into the special type, *MakeIntResource*. *MakeIntResource* is identical to the type *PChar*, but the name *MakeIntResource* makes your code clearer.

MaxCollectionSize variable — Objects

Declaration

```
MaxCollectionSize = 65520 div SizeOf(Pointer);
```

Function *MaxCollectionSize* determines the maximum number of elements that may be contained in a collection, which is essentially the number of pointers that can fit in a 64K memory segment.

mb_XXXX Message box flags — WinTypes

Function These flags specify characteristics of the message box created by the *MessageBox* function. You can combined them with **or** to create the desired style.

Values Windows defines the following flags:

Table 21.13
Message box flags

Constant	Meaning
mb_AbortRetryIgnore	Include Abort, Retry and Ignore buttons
mb_ApplModal	Create a modal message box (the default)
mb_DefButton1	The default button is the first button (the default)
mb_DefButton2	The default button is the second button
mb_DefButton3	The default button is the third button
mb_IconAsterisk	Same as *mb_IconInformation*
mb_IconExclamation	Include the '!' icon
mb_IconHand	Same as *mb_IconStop*
mb_IconInformation	Include the 'i' icon
mb_IconQuestion	Include the '?' icon
mb_IconStop	Include the stop sign icon
mb_Ok	Include OK button
mb_OkCancel	Include OK and Cancel buttons
mb_RetryCancel	Include Retry and Cancel buttons
mb_SystemModal	Create a modal message box that suspends Windows. Use this for potentially damaging situations.

Table 21.13: Message box flags (continued)

mb_TaskModal	Use this flag if no parent window is available. Supply 0 for the parent parameter, and Windows suspends all top-level windows in the application.
mb_YesNo	Include Yes and No buttons
mb_YesNoCancel	Include Yes, No, and Cancel buttons

There are several bit masks defined for groups of *mb_* constants:

Table 21.14 Message box flag masks

Mask	Constants masked
mb_DefMask	*mb_DefButton1, mb_DefButton2, mb_DefButton3*
mb_IconMask	*mb_IconAsterisk, mb_IconExclamation, mb_IconHand, mb_IconInformation, mb_IconQuestion, mb_IconStop*
mb_ModeMask	*mb_ApplModal, mb_SystemModal, mb_TaskModal*
mb_TypeMask	*mb_AbortRetryIgnore, mb_Ok, mb_OkCancel, mb_RetryCancel, mb_YesNo, mb_YesNoCancel*

See also *MessageBox* function

MemAlloc function — OMemory

Declaration

```
function MemAlloc(Size: Word): Pointer;
```

Function Allocates *Size* bytes of memory on the heap and returns a pointer to the block. If a block of the requested size cannot be allocated, a value of **nil** is returned. Unlike the *New* and *GetMem* standard procedures, *MemAlloc* will not allow the allocation to dip into the safety pool. A block allocated by *MemAlloc* can be disposed of using the *FreeMem* standard procedure.

MemAllocSeg function — OMemory

Declaration

```
function MemAllocSeg(Size: Word): Pointer;
```

Allocates a segment-aligned memory block. Corresponds to *MemAlloc*, except that the offset part of the resulting pointer is guaranteed to be zero.

See also *MemAlloc* function

nf_XXXX constants — OWindows

Function ObjectWindows defines several constants establishing ranges of notification messages.

Values The following constants are defined:

Table 21.15 Notification message constants

Constant	Value	Meaning
nf_First	$9000	Beginning of notification messages
nf_Count	$1000	Number of notification messages
nf_Internal	$9F00	Beginning of notification messages reserved for internal use

pf_XXXX constants — OPrinter

ObjectWindows defines a number of constants used to set flags in printout objects.

The following constants are defined:

Table 21.16 Printout flag constants

Constant	Value	Meaning
pf_Graphics	$01	Current band only accepts graphics
pf_Text	$02	Current band only accepts text
pf_Both	$03	Current band accepts both text and graphics
pf_Banding	$04	Set if the printout is being banded
pf_Selection	$08	Printing the selection

ps_XXXX constants — OPrinter

ObjectWindows defines several constants used by printer objects to determine printer status.

The following constants are defined:

Table 21.17 Printer state constants

Constant	Value	Meaning
ps_Ok	0	No error
ps_InvalidDevice	–1	Device parameters invalid
ps_Unassociated	–2	The printer object has no associated printer device

PString type — Objects

Declaration

```
PString = ^String;
```

Function Defines a pointer to a Pascal string.

PtrRec type — Objects

Declaration

```
PtrRec = record
  Ofs, Seg: Word;
end;
```

Function A record holding the offset and segment values of a pointer.

RegisterODialogs procedure — ODialogs

Declaration

```
procedure RegisterODialogs;
```

Function Calls *RegisterType* for each of the standard object types defined in the *ODialogs* unit: *TDialog*, *TDlgWindow*, *TControl*, *TButton*, *TCheckBox*, *TRadioButton*, *TGroupBox*, *TListBox*, *TComboBox*, *TScrollBar*, *TStatic*, and *TEdit*. After a call to *RegisterODialogs*, your application can read or write any of those types with streams.

See also *RegisterType* procedure

RegisterOStdWnds procedure — OStdWnds

Declaration

```
procedure RegisterOStdWnds;
```

Function Calls *RegisterType* for each of the standard object types defined in the *OStdWnds* unit: *TEditWindow* and *TFileWindow*. After a call to *RegisterOStdWnds*, your application can read or write any of those types with streams.

See also *RegisterType* procedure

RegisterOWindows procedure — OWindows

Declaration

```
procedure RegisterOWindows;
```

Function Calls *RegisterType* for each of the standard object types defined in the *OWindows* unit: *TWindow*, *TMDIWindow*, *TMDIClient*, and *TScroller*. After a call to *RegisterOWindows*, your application can read or write any of those types with streams.

See also *RegisterType* procedure

RegisterType procedure — Objects

Declaration `procedure RegisterType(var S: TStreamRec);`

Function A ObjectWindows object type must be registered using this method before it can be used in stream I/O. The standard object types are preregistered with *ObjType* values in the reserved range 0..99. *RegisterType* creates an entry in a linked list of *TStreamRec* records.

See also *TStream.Get*, *TStream.Put*, *TStreamRec*

RegisterValidate procedure — Validate

Declaration `procedure RegisterValidate;`

Function Calls *RegisterType* for each of the validator object types defined in the *Validate* unit: *TPXPictureValidator*, *TFilterValidator*, *TRangeValidator*, *TLookupValidator*, and *TStringLookupValidator*. After calling *RegisterValidate*, your application can read or write any of those types with streams.

See also *RegisterType* procedure

RestoreMemory procedure — OMemory

Declaration `procedure RestoreMemory;`

If *LowMemory* is *True*, meaning there is no safety pool, allocates *SafetyPoolSize* bytes to the safety pool.

See also *LowMemory* function, *SafetyPoolSize* variable

SafetyPoolSize variable — OMemory

Declaration `SafetyPoolSize: Word = 8192;`

Defines the size of the memory safety pool. The safety pool is a buffer at the high end of the heap used to ensure that memory allocations do not fail.

If you want to change the value of *SafetyPoolSize*, do so *before* calling *TApplication.Init*, which initializes the safety pool. Changing *SafetyPoolSize* after the application initializes the safety pool won't affect the size of the pool, but will cause the application to deallocate the wrong amount of memory.

See also *LowMemory, MemAlloc, TApplication.ValidWindow*

sbs_XXXX Scroll bar styles — WinTypes

Function You can pass these constants in the style parameter of scroll bar object constructors, or to specify scroll bar styles when creating scroll bars with *CreateWindow* or *CreateWindowEx*.

Values Windows defines the following styles:

Table 21.18
Scroll bar styles

Constant	Meaning
sbs_BottomAlign	The scroll bar is of default height and has its bottom edge aligned to the bottom edge of the rectangle used to create it. Use only with *sbs_Horz*.
sbs_Horz	The scroll bar is horizontal. If neither *sbs_BottomAlign* nor *sbs_TopAlign* is used, the scroll bar will be the exact size requested when it is created.
sbs_LeftAlign	The scroll bar is of default width and has its left edge aligned to the left edge of the rectangle used to create it. Use only with *sbs_Vert*.
sbs_RightAlign	The scroll bar is of default width and has its right edge aligned to the right edge of the rectangle used to create it. Use only with *sbs_Vert*.
sbs_SizeBox	The scroll bar is a size box. If neither *sbs_SizeBoxBottomRightAlign* nor *sbs_SizeBoxTopLeftAlign* is used, the scroll bar will be the exact size requested when it is created.
sbs_SizeBoxBottomRightAlign	The scroll bar is of default size for system size boxes and has its lower-right corner aligned to the lower-right corner of the rectangle used to create it. Use only with *sbs_SizeBox*.
sbs_SizeBoxTopLeftAlign	The scroll bar is of default size for system size boxes and has its upper-left corner aligned to the upper-left corner of the

Table 21.18: Scroll bar styles (continued)

	rectangle used to create it. Use only with *sbs_SizeBox*.
sbs_TopAlign	The scroll bar is of default height and has its top edge aligned to the top edge of the rectangle used to create it. Use only with *sbs_Horz*.
sbs_Vert	The scroll bar is vertical. If neither *sbs_RightAlign* nor *sbs_LeftAlign* is used, the scroll bar will be the exact size requested when it is created.

See also *TScrollBar.Init*

sd_XXXX constants — OStdDlgs

Function File dialog boxes use constants beginning with *sd_* to specify the resource template to use for constructing the dialog box. The program passes either *sd_FileOpen* or *sd_FileSave* to the file dialog box constructor, which then decides to use either normal or BWCC resource templates based on the value of *BWCCClassNames*.

Input dialog boxes also use *sd_XXXX* constants to specify their resources. Although the program doesn't need to specify a constant to the constructor, the input dialog box uses *sd_XXXX* constants internally to specify its resources.

Values The following constants are defined:

Table 21.19 Standard dialog box constants

Constant	Value	Meaning
sd_FileOpen	$7FFF	Use file open template
sd_FileSave	$7FFE	Use file save template
sd_WNFileOpen	$7F00	Normal file open template
sd_WNFileSave	$7F01	Normal file save template
sd_WNInputDialog	$7F02	Normal input dialog template
sd_BCFileOpen	$7F03	BWCC file open template
sd_BCFileSave	$7F04	BWCC file save template
sd_BCInputDialog	$7F05	BWCC input dialog template

See also *TFileDialog.Init, TInputDialog.Init*

ss_XXXX Static control styles — WinTypes

Function You can pass these constants in the style parameter of static control object constructors, or to specify static control styles when creating static controls with *CreateWindow* or *CreateWindowEx*.

Values Windows defines the following styles:

Table 21.20
Static control styles

Constant	Meaning
ss_BlackFrame	The static control has a frame with the same color as window frames.
ss_BlackRect	The static control is filled with the same color used to draw window frames.
ss_Center	The static control displays its text centered in its rectangle. If the text is longer than will fit in the width of the control, the line wraps to the next line. Lines break at word boundaries, with each line centered.
ss_GrayFrame	The static control has a frame with the same color as the screen background.
ss_GrayRect	The static control is filled with the color used to fill the screen background.
ss_Icon	The static control is an icon. The text of the control is the name of the icon resource. Icons automatically size themselves.
ss_Left	The static control displays its text flush left in its rectangle. If the text is longer than will fit in the width of the control, the line wraps to the next line. Lines break at word boundaries, with each line left aligned.
ss_LeftNoWordWrap	The static control displays its text flush left in its rectangle. If the text is longer than will fit in the width of the control, the extra text is clipped.
ss_NoPrefix	The static control ignores '&' characters in its text. Normally '&' is an accelerator prefix character which is removed and the next character in the string is underlined.
ss_Right	The static control displays its text flush right in its rectangle. If the text is longer than will fit in the width of the control, the text wraps to the next line. Lines break at word boundaries, with each line right aligned.
ss_Simple	The static control displays one line of text flush left. The text cannot be altered. The control's parent must not process the *wm_CtlColor* message.
ss_UserItem	The static control is a user-defined static control.
ss_WhiteFrame	The static control has a frame with the same color as window backgrounds.
ss_WhiteRect	The static control is filled with the color used to fill window backgrounds.

See also *TStatic.Init*

Stock logical objects — WinTypes

Function These constants represent predefined (stock) drawing tools. Pass them to the *GetStockObject* API function to return a handle to the specified object.

Values Windows defines the following stock logical objects:

Table 21.21 Stock logical object constants

Constant	Meaning
Black_Brush	Black brush
DkGray_Brush	Dark gray brush
Gray_Brush	Gray brush
Hollow_Brush	Hollow brush
LtGray_Brush	Light gray brush
Null_Brush	Null brush
White_Brush	White brush
Black_Pen	Black pen
Null_Pen	Null pen
White_Pen	White pen
ANSI_Fixed_Font	ANSI fixed-pitch system font
ANSI_Var_Font	ANSI variable-pitch system font
Device_Default_Font	Device-dependent font
OEM_Fixed_Font	OEM-dependent fixed font
System_Fixed_Font	Fixed-pitch font from previous versions of Windows
System_Font	The Windows system font (variable-pitch)
Default_Palette	The default color palette

StreamError variable — Objects

Declaration

```
StreamError: Pointer = nil;
```

Function If non-**nil**, *StreamError* points to a procedure that will be called by a stream's *Error* method when a stream error occurs. The procedure must be a **far** procedure with one **var** parameter that is a *TStream*. That is, the procedure must be declared as

```
procedure MyStreamErrorProc(var S: TStream); far;
```

StreamError allows you to globally override all stream error handling. To change error handling for a particular type of stream, override that stream type's *Error* method.

stXXXX constants — Objects

Function There are two sets of constants beginning with "st" that are used by the ObjectWindows streams system.

Values The following mode constants are used by *TDosStream* and *TBufStream* to determine the file access mode of a file being opened for an ObjectWindows stream:

Table 21.22 Stream access modes

Constant	Value	Meaning
stCreate	$3C00	Create new file
stOpenRead	$3D00	Open existing file with read access only
stOpenWrite	$3D01	Open existing file with write access only
stOpen	$3D02	Open existing file with read and write access

The following values are returned by *TStream.Error* in the *TStream.ErrorInfo* field when a stream error occurs:

Table 21.23 Stream error codes

Error code	Value	Meaning
stOk	0	No error
stError	–1	Access error
stInitError	–2	Cannot initialize stream
stReadError	–3	Read beyond end of stream
stWriteError	–4	Cannot expand stream
stGetError	–5	Get of unregistered object type
stPutError	–6	Put of unregistered object type

See also *TStream* object

sw_XXXX Show window constants — WinTypes

Function These constants indicate the state in which the *ShowWindow* function displays a window. The *TWindowsObject* method *Show* takes one of these constants as a parameter and passes it to *ShowWindow*.

You can use *sw_* constants to set the initial state of an ObjectWindows application's main window by setting the value of the *System* unit variable *CmdShow* before constructing the main window.

Values Windows defines the following constants:

Table 21.24 ShowWindow constants

Constant	Meaning
sw_Hide	Hidden
sw_Maximize	Same as *sw_ShowMaximized*
sw_Minimize	Minimized and inactive

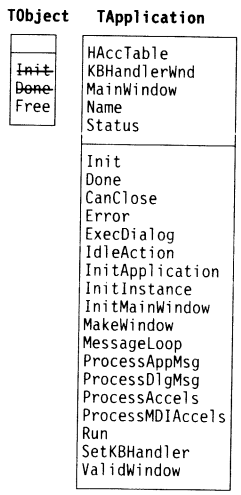
Table 21.24: ShowWindow constants (continued)

sw_Normal	Same as *sw_ShowNormal*
sw_OtherZoom	Another window is being maximized (Included for Windows 2.0 compatibility)
sw_OtherUnzoom	Another window is being minimized (Included for Windows 2.0 compatibility)
sw_Restore	Same as *sw_ShowNormal*
sw_Show	In the window's current size and position
sw_ShowMaximized	Maximized and active
sw_ShowMinimized	Minimized and active
sw_ShowMinNoActive	Minimized; does not affect window activation
sw_ShowNA	In the window's current state; does not affect window activation
sw_ShowNoActivate	In the window's current size and position; does not affect window activation
sw_ShowNormal	Restored and active

See also *CmdShow* variable, *TWindowsObject.Show*

TApplication

OWindows



TApplication provides the structure for an ObjectWindows applications. All ObjectWindows applications derive an object type from *TApplication*, primarily to construct a main window of a user-defined object type.

Fields

HAccTable

```
HAccTable: THandle;
```

Read/write

HAccTable holds a handle to a Windows accelerator table resource defined for the application.

KBHandlerWnd

```
KBHandlerWnd: PWindowsObject;
```

Read only

KBHandlerWnd points to the currently active window if that window's keyboard handler mechanism is enabled. This mechanism allows a window with controls to process keyboard input like a dialog. *KBHandlerWnd* is **nil** if the mechanism is disabled for the active window.

MainWindow

```
MainWindow: PWindowsObject;
```

Read/write

MainWindow points to the application's main, overlapped window, which should be instantiated by your application type's *InitMainWindow* method.

Name

```
Name: PChar;
```

Name holds the name of the application.

Status

```
Status: Integer;
```

Status indicates the current state of the running application. It is running successfully if *Status* is greater than or equal to zero. Error values include *em_InvalidMainWindow* for an invalid main window object.

Methods

Init

```
constructor Init(AName: PChar);
```

Override: Sometimes

Constructs the application object by first calling the *Init* constructor inherited from *TObject*, then setting the *Application* variable to *@Self*, setting *Name* to *AName* and *HAccTable* and *Status* to zero, and finally setting *MainWindow* and *KBHandlerWnd* to **nil**.

If this is the first running instance of this application, *Init* calls *InitApplication*. If *InitApplication* succeeds (that is, the *Status* field is still zero), *InitInstance* is called.

You may override this method to, for example, load an accelerator table for your application. Make sure you call this method from any method which overrides it.

See also: *TApplication.InitApplication*, *TApplication.InitInstance*, *TObject.Init*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Disposes of the application's owned objects by disposing of *MainWindow*, then calling the *Done* destructor inherited from *TObject* to terminate the application.

See also: *TObject.Done*

CanClose

```
function CanClose: Boolean; virtual;
```

Override: Seldom

Returns *True* if it is OK for the application to close. As a default, it calls the *CanClose* method of its main window and returns its return value. This method will seldom be overridden; closing behavior can be overridden in the main window's *CanClose* method.

See also: *TWindowsObject.CanClose, TWindowsObject.WMDestroy*

Error

```
procedure Error(ErrorCode: Integer); virtual;
```

Override: Often

Error processes errors identified by the error value passed in *ErrorCode*. These errors can be generated by the application object or any window or dialog object, and *ErrorCode* can be one of the following errors detected and reported by ObjectWindows, or an error you define:

em_InvalidWindow
em_OutOfMemory
em_InvalidClient
em_InvalidChild
em_InvalidMainWindow

The *em_XXXX* constants are described in this chapter.

Error displays the error code in a message box and asks the user if it is OK to proceed. If not, program execution stops.

ExecDialog

```
function ExecDialog(ADialog: PWindowsObject): Integer; virtual;
```

Override: Never

Executes the modal dialog object passed in *ADialog*, after checking *ValidWindow*, by calling the dialog object's *Execute* method. If memory is low, or if the dialog cannot be executed, *ExecDialog* disposes of the object and returns a negative error status.

See also: *TDialog.Execute*

IdleAction

```
function IdleAction: Boolean;
```

IdleAction provides an opportunity for your application to perform background processing outside the message loop. Whenever *MessageLoop* determines that no messages are pending for the application, it calls *IdleAction*, which can perform incremental background processing.

If *IdleAction* returns *True*, the message loop continues to call *IdleAction* for further processing while still waiting for Windows messages. If *IdleAction* returns *False*, the message loop *only* waits for messages from Windows, without calling *IdleAction* again.

After the application receives and processes a message from Windows and again goes idle, it will call *IdleAction* again.

By default, *IdleAction* always returns *False*.

Actions performed by *IdleAction* should either be short, complete actions or short, incremental parts of a larger action. Otherwise, the application will respond sluggishly to user and system messages.

See also: *TApplication.MessageLoop*

InitApplication

```
procedure InitApplication; virtual;
```

Override: Sometimes

Performs any initialization necessary only for the first executing instance of the application. *TApplication.InitApplication* does nothing. Your application object type can override *InitApplication* to perform application-specific initialization.

InitInstance

```
procedure InitInstance; virtual;
```

Override: Sometimes

Performs any initialization necessary for every executing instance of the application. *TApplication.InitInstance* calls *InitMainWindow* and creates and shows the main window element by calling *MakeWindow* and *Show*. If the main window could not be created, the *Status* field is set to *em_InvalidMainWindow*. If you override this method, be sure to explicitly call *TApplication.InitInstance*.

See also: *TApplication.InitMainWindow*

InitMainWindow

```
procedure InitMainWindow; virtual;
```

Override: Always

By default, *TApplication.InitMainWindow* instantiates a generic *TWindow* object with no title. Override *InitMainWindow* to construct a useful main window object and store it in *MainWindow*. Typical use:

```
procedure MyApplication.InitMainWindow;
begin
  MainWindow := New(PMyWindow, Init(nil, 'Window Caption'));
end;
```

MakeWindow

```
function MakeWindow(AWindowsObject: PWindowsObject): PWindowsObject; virtual;
```

Override: Never

Attempts to create a window or modeless dialog element associated with the object passed in *AWindowsObject*, after performing checks of safety

pool usage. If memory is low (*LowMemory* returns *True*), or if the window or dialog cannot be created, *MakeWindow* disposes the object and returns **nil**. If successful, it returns *AWindowsObject*.

See also: *TWindow.Create, LowMemory*

MessageLoop

```
procedure MessageLoop; virtual;
```

Override: Never

Operates the application's general message loop which runs during the lifetime of the application. *MessageLoop* calls *ProcessAppMsg* to handle special messages for modeless dialog boxes, accelerators and MDI accelerators. Any nonstandard message processing should be done in *ProcessAppMsg*, rather than in *MessageLoop*. If *MessageLoop* determines that no messages are pending for the application, it calls *IdleAction* to run any background processes.

See also: *TApplication.IdleAction, TApplication.ProcessAppMsg*

ProcessAccels

```
function ProcessAccels(var Message: TMsg): Boolean; virtual;
```

Override: Sometimes

Handles special accelerator message processing. If your application's windows do not respond to accelerators, you can improve performance by overriding this method to simply return *False*.

ProcessAppMsg

```
function ProcessAppMsg(var Message: TMsg): Boolean; virtual;
```

Override: Sometimes

Checks for special processing for modeless dialog box, accelerator, and MDI accelerator messages. Calls *ProcessDlgMsg*, *ProcessMDIAccels*, and *ProcessAccels* and returns *True* if any of these special messages are encountered. If your application does not create modeless dialogs, does not respond to accelerators, and is not an MDI application, you can improve performance by overriding this method to simply return *False*.

ProcessDlgMsg

```
function ProcessDlgMsg(var Message: TMsg): Boolean; virtual;
```

Override: Sometimes

Handles special modeless dialog and window message processing for handling keyboard input for controls. If your application creates no modeless dialogs or windows with controls, you can improve performance by overriding this method to simply return *False*.

ProcessMDIAccels

```
function ProcessMDIAccels(var Message: TMsg): Boolean; virtual;
```

Handles special accelerator message processing for MDI-compliant applications. If your application is not an MDI application, you can improve performance by overriding this method to simply return *False*.

Run

```
procedure Run; virtual;
```

Override: Seldom

Sets the application in motion by calling *MessageLoop* if initialization was successful (that is, the *Status* field is zero).

See also: *TApplication.MessageLoop*

SetKBHandler

```
procedure SetKBHandler(AWindowsObject: PWindowsObject);
```

Override: Never

Activates keyboard handling (translation of keyboard input into control selections) for the given window by setting *KBHandlerWnd* to *AWindowsObject*.

See also: *TApplication.KBHandlerWnd*

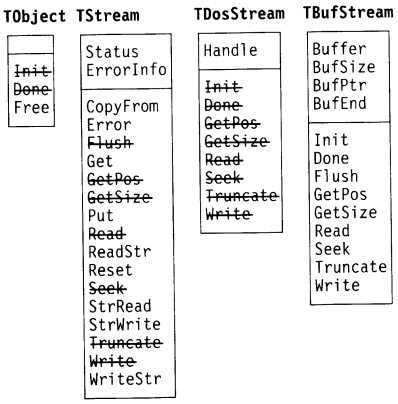
ValidWindow

```
function ValidWindow(AWindowsObject: PWindowsObject): PWindowsObject;
```

Determines whether *AWindowsObject* is a valid object. If *AWindowsObject* is valid, *ValidWindow* returns a pointer to it, otherwise, it returns **nil**. *AWindowsObject* is invalid if either of two conditions occurs: allocation of the object ate into the safety pool (*LowMemory* is *True*), or the *Status* field of *AWindowsObject* is nonzero.

See also: *LowMemory, TWindowsObject.Status*

TBufStream



TBufStream implements a buffered version of *TDosStream*. The additional fields specify the size and location of the buffer, together with the current and last positions within the buffer. In addition to overriding the eight methods of *TDosStream*, *TBufStream* defines the abstract *TStream.Flush* method. The *TBufStream* constructor creates and opens a named file by calling *TDosStream.Init*, then creates the buffer with *GetMem*.

TBufStream is significantly more efficient than *TDosStream* when a large number of small data transfers take place on the stream, such as when loading and storing objects using *TStream.Get* and *TStream.Put*.

Fields

BufEnd

```
BufEnd: Word;
```

Read only

If the buffer is not full, *BufEnd* gives an offset from the *Buffer* pointer to the last used byte in the buffer.

Buffer

```
Buffer: Pointer;
```

Read only

Points to the start of the stream's buffer.

BufPtr

```
BufPtr: Word;
```

Read only

An offset from the *Buffer* pointer indicating the current position within the buffer.

BufSize

```
BufSize: Word;
```

Read only

The size of the buffer in bytes.

Methods

Init

```
constructor Init(FileName: FNameStr; Mode, Size: Word);
```

Creates and opens the named file with access mode *Mode* by calling *TDosStream.Init*. Also creates a buffer of *Size* bytes with a *GetMem* call. The *Handle*, *Buffer* and *BufSize* fields are suitably initialized. Typical buffer sizes range from 512 bytes to 2,048 bytes.

See also: *TDosStream.Init*

Done

```
destructor Done; virtual;
```

Override: Never

Closes and disposes of the file stream; flushes and disposes of its buffer.

See also: *TBufStream.Flush*

Flush

```
procedure Flush; virtual;
```

Override: Never

Flushes the stream's buffer provided the stream is *stOK*.

See also: *TBufStream.Done*

GetPos

```
function GetPos: Longint; virtual;
```

Override: Never

Returns the value of the stream's current position (not to be confused with *BufPtr*, the current location within the buffer).

See also: *TBufStream.Seek*

GetSize `function GetSize: Longint; virtual;`

Override: Never

Flushes the buffer, then returns the total size in bytes of the stream.

Read `procedure Read(var Buf; Count: Word); virtual;`

Override: Never

If *stOK*, reads *Count* bytes into the *Buf* buffer starting at the stream's current position.

Note that *Buf* is *not* the stream's buffer, but an external buffer to hold the data read in from the stream.

See also: *stReadError*, *TBufStream.Write*

Seek `procedure Seek(Pos: Longint); virtual;`

Override: Never

Flushes the buffer, then resets the current position to *Pos* bytes from the start of the stream. The start of a stream is position 0.

See also: *TBufStream.GetPos*

Truncate `procedure Truncate; virtual;`

Override: Never

Flushes the buffer, then deletes all data on the stream from the current position to the end. The current position is set to the new end of the stream.

See also: *TBufStream.GetPos*, *TBufStream.Seek*

Write `procedure Write(var Buf; Count: Word); virtual;`

Override: Never

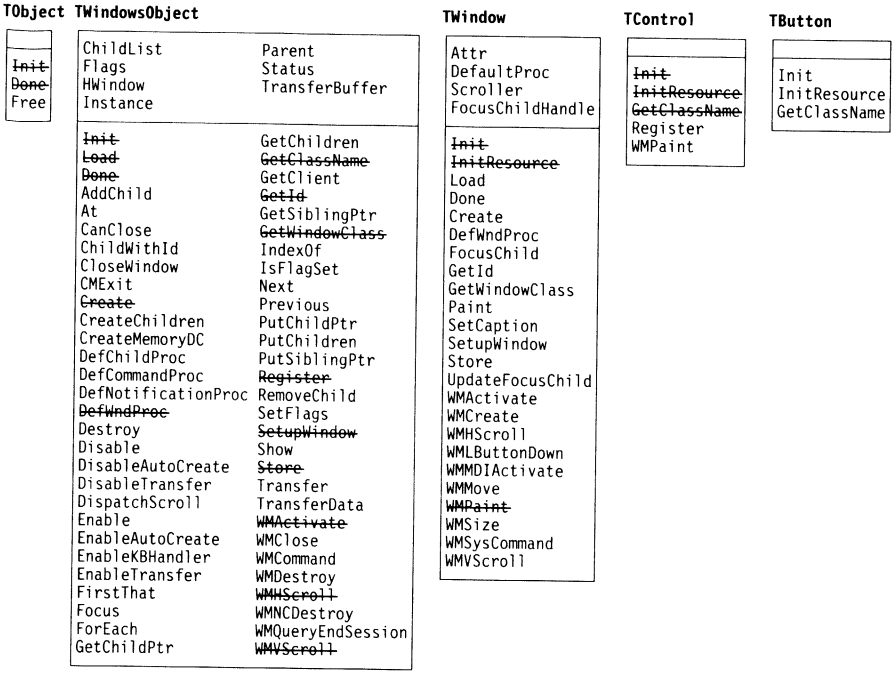
If *stOK*, writes *Count* bytes from the *Buf* buffer to the stream, starting at the current position.

Note that *Buf* is *not* the stream's buffer, but an external buffer to hold the data being written to the stream. When *Write* is called, *Buf* will point to the variable whose value is being written.

See also: *stWriteError*, *TBufStream.Read*

TButton

ODialogs



TButton is an interface object that represents a corresponding push button element in Windows. There are two types of push buttons. A regular button appears with a thin border. A default button appears with a thick border and represents the default action of the window. There can only be one default push button in a window.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnId: Integer; AText: PChar;
  X, Y, W, H: Integer; IsDefault: Boolean);
```

Constructs a button object with the passed parent window (*AParent*), control ID (*AnId*), associated text (*AText*), position relative to the origin of the parent window's client area (*X*, *Y*), width (*W*), and height (*H*). Calls *TControl.Init* and then adds *bs_DefPushButton* to the *Attr.Style* field if *IsDefault* is *True*, else adds *bs_PushButton*.

See also: *TControl.Init*

InitResource

```
constructor InitResource(AParent: PWindowsObject, ResourceID: Word);
```

Associates the button by constructing an ObjectWindows object to correspond to a button element created by a dialog resource definition. Calls *TControl.InitResource* and *DisableTransfer* to exclude the button from the transfer mechanism, since they have no data to be transferred.

See also: *TControl.InitResource, TWindowsObject.DisableTransfer*

GetClassName `function GetClassName: PChar; virtual;`

Override: Never

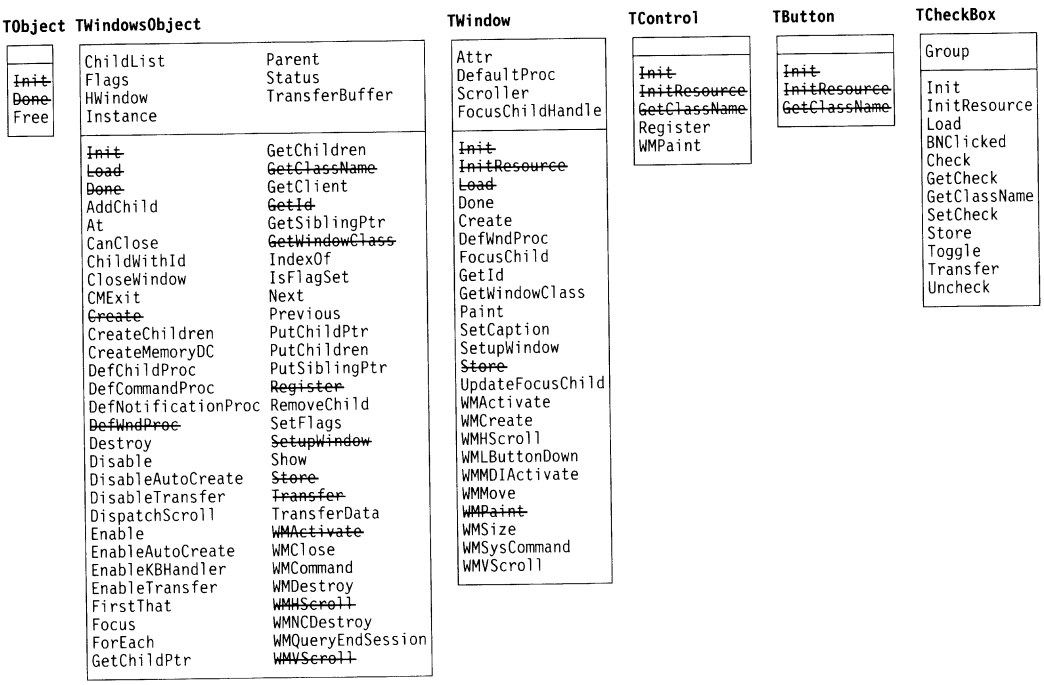
Returns the name of *TButton's* window class, 'Button'. If you're using BWCC controls, the class is 'BorBtn'.

TByteArray type — Objects

Declaration `TByteArray = array[0..32767] of Byte;`

Function A byte array type for general use in typecasts.

TCheckBox — ODialogs



TCheckBox is an interface object that represents a corresponding check box element in Windows. Check boxes have two states: checked and unchecked. *TCheckBox* methods are concerned primarily with managing the check box's state. Optionally, a check box can be part of a group (*TGroupBox*) which visually and functionally groups its controls.

Field

Group

```
Group: PGroupBox;                                    Read only
```

Group points to the *TGroupBox* control object that unifies the check box with other check boxes and radio buttons (*TRadioButton*). If the check box is not part of a group, *Group* is equal to **nil**.

See also: *TGroupBox*, *TRadioButton*

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; ATitle:
  PChar; X, Y, W, H: Integer; AGroup: PGroupBox);
```

Override: Sometimes

Constructs a check box object with the passed parent window (*AParent*), control ID (*AnId*), associated text (*ATitle*), position relative to the origin of the parent window's client area (*X*, *Y*), width (*W*), height (*H*), and associated group box (*AGroup*). *TCheckBox.Init* sets the check box's *Attr.Style* field to *ws_Child* **or** *ws_Visible* **or** *ws_TabStop* **or** *bs_AutoCheckBox*.

InitResource

```
constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
```

Associates a *TCheckBox* object with the resource given by *ResourceID* by calling the *InitResource* constructor inherited from *TButton*, then enables the transfer mechanism by calling *EnableTransfer*.

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a check box from the stream *S* by first calling the *Load* constructor inherited from *TButton* and then reading the additional field (*Group*) introduced by *TCheckBox*.

See also: *TControl.Load*

BNClicked

```
procedure BNClicked(var Msg: TMessage); virtual nf_First + bn_Clicked;
```

Override: Sometimes

Automatically responds to notification messages indicating that the check box was clicked by toggling its state. If the check box's *Group* is not **nil**, *TCheckBox.BNClicked* notifies the *TGroupBox* by calling its *SelectionChanged* method.

See also: *TGroupBox.SelectionChanged*

Check

```
procedure Check; virtual;
```

Override: Seldom

Forces the check box into the checked state by calling *SetCheck*.

See also: *TCheckBox.SetCheck*

GetCheck

```
function GetCheck: Word; virtual;
```

Override: Seldom

Returns *bf_Unchecked* if the check box is unchecked, *bf_Checked* if it is checked, or *bf_Grayed* if it is grayed.

GetClassName

```
function GetClassName: PChar; virtual;
```

Calls the *GetClassName* method inherited from *TButton* unless you're using BWCC controls, in which case it returns 'BorCheck'.

SetCheck

```
procedure SetCheck(CheckFlag: Word); virtual;
```

Override: Seldom

Forces the check box into the state specified by *CheckFlag*. If *CheckFlag* is *bf_Unchecked*, the state will be unchecked. If it is *bf_Checked*, the state will be checked. If it is *bf_Grayed*, the state will be grayed. *SetCheck* also informs the check box's group that the selection has changed.

See also: *TGroupBox.SelectionChanged*

Store

```
procedure Store(var S: TStream);
```

Stores the check box on the stream *S* by first calling *TControl.Store* and then writing the additional field (*Group*) introduced by *TCheckBox*.

See also: *TControl.Store*

Toggle

```
procedure Toggle; virtual;
```

Override: Seldom

Toggles the state of the check box by calling *Check* or *Uncheck*. For a three-state check box, toggles through all three states: checked, unchecked, and grayed.

See also: *TCheckBox.Check*, *TCheckBox.Uncheck*

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word;
  virtual;
```

Override: Sometimes

Transfers the state of the check box in a *Word*-type value (*bf_Checked* if checked, *bf_Unchecked* if unchecked) to or from the memory location pointed to by *DataPtr*. If *TransferFlag* is *tf_GetData*, the check box's state data is transferred to the memory location. If *TransferFlag* is *tf_SetData*, the check box is set to the state indicated in the memory location. *Transfer* returns the number of bytes stored in or retrieved from the memory location. If you pass *tf_SizeData*, *Transfer* returns the size of the transfer data, which is two bytes.

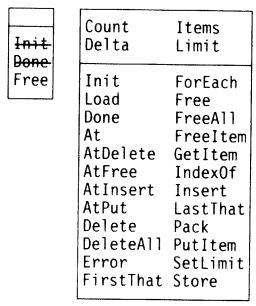
Uncheck `procedure Uncheck; virtual;`

Override: Seldom Forces the check box into the unchecked state by calling *SetCheck*.

See also: *TCheckBox.SetCheck*

TCollection Objects

TObject TCollection



TCollection is an abstract type for implementing any collection of items, including other objects. *TCollection* is a more general concept than the traditional array, set, or list. *TCollection* objects size themselves dynamically at run time and offer a base type for many specialized types such as *TSortedCollection* and *TStrCollection*. In addition to methods for adding and deleting items, *TCollection* offers several *iterator* routines that call a procedure or function for each item in the collection.

Fields

Count `Count: Integer;` **Read only**

The current number of items in the collection, up to *MaxCollectionSize*.

See also: *MaxCollectionSize* variable

Delta `Delta: Integer;` **Read only**

The number of items by which to increase the *Items* list whenever it becomes full. If *Delta* is zero, the collection cannot grow beyond the size set by *Limit*.

Increasing the size of a collection is fairly costly in terms of performance. To minimize the number of times it has to occur, try to set the initial *Limit* to an amount that will encompass all the items you might want to collect, and set *Delta* to a figure that will allow a reasonable amount of expansion.

See also: *Limit, TCollection.Init*

Items `Items: PItemList;` **Read only**

A pointer to an array of item pointers.

See also: *TItemList* type

Limit `Limit: Integer;` **Read only**

The currently allocated size (in elements) of the *Items* list.

See also: *Delta, TCollection.Init*

Methods

Init `constructor Init(ALimit, ADelta: Integer);`

Creates a collection with *Limit* set to *ALimit* and *Delta* set to *ADelta*. The initial number of items is limited to *ALimit*, but the collection can grow in increments of *ADelta* until memory runs out or the number of items reaches *MaxCollectionSize*.

See also: *TCollection.Limit, TCollection.Delta*

Load `constructor Load(var S: TStream);`

Creates and loads a collection from the given stream. *Load* calls *GetItem* for each item in the collection.

See also: *TCollection.GetItem*

Done `destructor Done; virtual;`

Override: Often

Deletes and disposes of all items in the collection by calling *FreeAll* and setting *Limit* to 0.

See also: *TCollection.FreeAll, TCollection.Init*

At `function At(Index: Integer): Pointer;`

Returns a pointer to the item indexed by *Index* in the collection. This method lets you treat a collection as an indexed array. If *Index* is less than zero or greater than or equal to *Count*, the *Error* method is called with an argument of *coIndexError*, and a value of **nil** is returned.

See also: *TCollection.IndexOf*

AtDelete `procedure AtDelete(Index: Integer);`

Deletes the item at the *Index*'th position and moves the following items up by one position. *Count* is decremented by 1, but the memory allocated to

the collection (as given by *Limit*) is not reduced. If *Index* is less than zero or greater than or equal to *Count*, the *Error* method is called with an argument of *coIndexError*.

See also: *TCollection.FreeItem, TCollection.Free, TCollection.Delete*

AtFree

```
procedure AtFree(Index: Integer);
```

Disposes of and deletes the item at the *Index*'th position.

AtInsert

```
procedure AtInsert(Index: Integer; Item: Pointer);
```

Inserts *Item* at the *Index*'th position and moves the following items down by one position. If *Index* is less than zero or greater than *Count*, the *Error* method is called with an argument of *coIndexError* and the new *Item* is not inserted. If *Count* is equal to *Limit* before the call to *AtInsert*, the allocated size of the collection is expanded by *Delta* items using a call to *SetLimit*. If the *SetLimit* call fails to expand the collection, the *Error* method is called with an argument of *coOverflow* and the new *Item* is not inserted.

See also: *TCollection.At, TCollection.AtPut*

AtPut

```
procedure AtPut(Index: Integer; Item: Pointer);
```

Replaces the item at index position *Index* with the item given by *Item*. If *Index* is less than zero or greater than or equal to *Count*, the *Error* method is called with an argument of *coIndexError*.

See also: *TCollection.At, TCollection.AtInsert*

Delete

```
procedure Delete(Item: Pointer);
```

Deletes the item given by *Item* from the collection. Equivalent to *AtDelete(IndexOf(Item))*.

See also: *TCollection.AtDelete, TCollection.DeleteAll*

DeleteAll

```
procedure DeleteAll;
```

Deletes all items from the collection by setting *Count* to zero.

See also: *TCollection.Delete, TCollection.AtDelete*

Error

```
procedure Error(Code, Info: Integer); virtual;
```

Override: Sometimes

Called whenever a collection error is encountered. By default, this method produces a run-time error of (212 – *Code*).

See also: *coXXXX* collection constants

FirstThat

```
function FirstThat(Test: Pointer): Pointer;
```

FirstThat applies a Boolean function, given by the function pointer *Test*, to each item in the collection until *Test* returns *True*. The result is the item pointer for which *Test* returned *True*, or **nil** if the *Test* function returned *False* for all items. *Test* must point to a **far** local function taking one *Pointer* parameter and returning a *Boolean* value. For example

```
function Matches(Item: Pointer): Boolean; far;
```

The *Test* function *cannot* be a global function.

Assuming that *List* is a *TCollection*, the statement

```
P := List.FirstThat(@Matches);
```

corresponds to

```
I := 0;
while (I < List.Count) and not Matches(List.At(I)) do Inc(I);
if I < List.Count then P := List.At(I) else P := nil;
```

See also: *TCollection.LastThat*, *TCollection.ForEach*

ForEach

```
procedure ForEach(Action: Pointer);
```

ForEach applies an action, given by the procedure pointer *Action*, to each item in the collection. *Action* must point to a **far** local procedure taking one *Pointer* parameter. For example

```
procedure PrintItem(Item: Pointer); far;
```

The *Action* procedure *cannot* be a global procedure.

Assuming that *List* is a *TCollection*, the statement

```
List.ForEach(@PrintItem);
```

corresponds to

```
for I := 0 to List.Count - 1 do PrintItem(List.At(I));
```

See also: *TCollection.FirstThat*, *TCollection.LastThat*

Free

```
procedure Free(Item: Pointer);
```

Deletes and disposes of the given *Item*. Equivalent to

```
Delete(Item);
FreeItem(Item);
```

See also: *TCollection.FreeItem*, *TCollection.Delete*

FreeAll

```
procedure FreeAll;
```

Deletes and disposes of all items in the collection.

See also: *TCollection.DeleteAll*

FreeItem

```
procedure FreeItem(Item: Pointer); virtual;
```

Override: Sometimes

The *FreeItem* method must dispose of the given *Item*. The default *FreeItem* assumes that *Item* is a pointer to a descendant of *TObject*, and thus calls the *Done* destructor:

```
if Item <> nil then Dispose(PObject(Item), Done);
```

FreeItem is called by *Free* and *FreeAll*, but it should never be called directly.

See also: *TCollection.Free, TCollection.FreeAll*

GetItem

```
function TCollection.GetItem(var S: TStream): Pointer; virtual;
```

Override: Sometimes

Called by *Load* for each item in the collection. This method can be overridden but should not be called directly. The default *GetItem* assumes that the items in the collection are descendants of *TObject*, and thus calls *TStream.Get* to load the item:

```
GetItem := S.Get;
```

See also: *TStream.Get, TCollection.Load, TCollection.Store*

IndexOf

```
function IndexOf(Item: Pointer): Integer; virtual;
```

Override: Never

Returns the index of the given *Item*. The converse operation to *TCollection.At*. If *Item* is not in the collection, *IndexOf* returns –1.

See also: *TCollection.At*

Insert

```
procedure Insert(Item: Pointer); virtual;
```

Override: Never

Inserts *Item* into the collection and adjusts other indexes if necessary. By default, insertions are made at the end of the collection by calling *AtInsert(Count, Item);*

See also: *TCollection.AtInsert*

LastThat

```
function LastThat(Test: Pointer): Pointer;
```

LastThat applies a Boolean function, given by the function pointer *Test*, to each item in the collection in reverse order until *Test* returns *True*. The result is the item pointer for which *Test* returned *True*, or **nil** if the *Test* function returned *False* for all items. *Test* must point to a **far** local function taking one *Pointer* parameter and returning a *Boolean* value. For example:

```
function Matches(Item: Pointer): Boolean; far;
```

The *Test* function *cannot* be a global function.

Assuming that *List* is a *TCollection*, the statement

```
P := List.LastThat(@Matches);
```

corresponds to

```
I := List.Count - 1;
while (I >= 0) and not Matches(List.At(I)) do Dec(I);
if I >= 0 then P := List.At(I) else P := nil;
```

See also: *TCollection.FirstThat*, *TCollection.ForEach*

Pack

```
procedure Pack;
```

Deletes all **nil** pointers in the collection.

See also: *TCollection.Delete*, *TCollection.DeleteAll*

PutItem

```
procedure PutItem(var S: TStream; Item: Pointer); virtual;
```

Override: Sometimes

Called by *TCollection.Store* for each item in the collection. This method can be overridden but should not be called directly. The default *TCollection.PutItem* assumes that the items in the collection are descendants of *TObject*, and thus calls *TStream.Put* to store the item:

```
S.Put(Item);
```

See also: *TCollection.GetItem*, *TCollection.Store*, *TCollection.Load*

SetLimit

```
procedure SetLimit(ALimit: Integer); virtual;
```

Override: Seldom

Expands or shrinks the collection by changing the allocated size to *ALimit*. If *ALimit* is less than *Count*, it is set to *Count*, and if *ALimit* is greater than *MaxCollectionSize*, it is set to *MaxCollectionSize*. Then, if *ALimit* is different from the current *Limit*, a new *Items* array of *ALimit* elements is allocated, the old *Items* array is copied into the new array, and the old array is disposed of.

See also: *TCollection.Limit*, *TCollection.Count*, *MaxCollectionSize* variable

Store

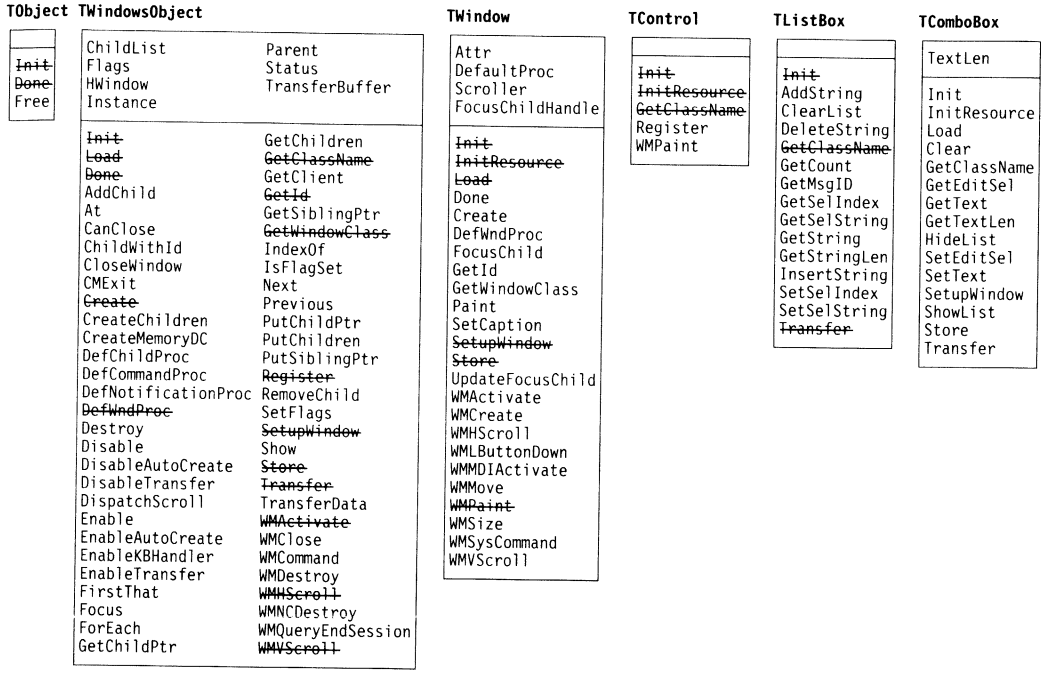
```
procedure Store(var S: TStream);
```

Stores the collection and all its items on the stream *S*. *TCollection.Store* calls *TCollection.PutItem* for each item in the collection.

See also: *TCollection.PutItem*

TComboBox

ODialogs



TComboBox is an interface object that represents a corresponding combo box element in Windows. Combo box objects inherit most of their functionality from *TListBox*.

There are three types of combo boxes: simple, drop down, and drop down list. These types are governed by Windows style constants: *cbs_Simple*, *cbs_DropDown*, and *cbs_DropDownList*. These constants are passed to the *Init* constructor, which in turn tells Windows which type of combo box element to create.

In this version of ObjectWindows, combo boxes have several new methods for manipulating the edit portion of the control.

Field

TextLen

```
TextLen: Word;
```

Read only

TextLen contains the length of the character buffer in the edit portion of the combo box, which is also the number of bytes transferred by *Transfer*. *TextLen* is set by *Init*.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; X, Y, W, H:
  Integer; AStyle, ATextLen: Word);
```

Override: Sometimes

Constructs a combo box object with the passed parent window (*AParent*), control ID (*AnId*), position relative to the origin of the parent window's client area (*X*, *Y*), width (*W*), and height (*H*), by calling the *Init* constructor inherited from *TListBox*. Sets *TextLen* to *ATextLen*. Sets *Attr.Style* to (*Attr.Style* **and not** *lbs_Notify*) **or** *AStyle* **or** *cbs_Sort* **or** *ws_VScroll* **or** *ws_HScroll*.

See also: *TListBox.Init*, *cbs_* Combo box style constants

InitResource

```
constructor InitResource(AParent:PWindowsObject; ResourceID: Integer;
  ATextLen: Word);
```

Associates a *TComboBox* object with the resource indicated by *ResourceID*, with a maximum text length of *ATextLen* – 1.

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a combo box from the stream *S* by first calling *TListBox.Load* and then reading the additional fields (*Style*, *TextLen*) introduced by *TComboBox*.

See also: *TListBox.Load*

Clear

```
procedure Clear;
```

Clears the text in the combo box's edit portion by calling *SetText('')*.

See also: *TComboBox.SetText*

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Never

Returns the name of *TComboBox's* window class, 'ComboBox'.

GetEditSel

```
function GetEditSel(var StartPos, EndPos: Integer): Boolean;
```

Sets *StartPos* and *EndPos* to the starting and ending positions of the selected text in the combo box's edit portion. Returns *False* if the combo box has no edit control; otherwise, returns *True*.

GetText

```
function GetText(Str: PChar; MaxChars: Integer): Integer;
```

Sets *Str* to the text of the associated edit control, up to a maximum of *MaxChars* characters and returns the number of characters copied.

GetTextLen

```
function GetTextLen: Integer;
```

Returns the length of the text in the associated edit control.

HideList

```
procedure HideList;
```

Forces the hiding of the drop down list of a drop down or drop down list combo box.

SetEditSel

```
function SetEditSel(StartPos, EndPos: Integer): Integer;
```

Selects text in the combo box's edit control between the positions *StartPos* and *EndPos*. Returns *cb_Err* if the combo box has no edit control; otherwise, returns zero.

SetText

```
procedure SetText(Str: PChar);
```

Sets the text in the combo box's edit control to *Str*.

SetupWindow

```
procedure SetupWindow;
```

Initializes the combo box object by first calling the *SetupWindow* method inherited from *TListBox*, then sending a *cb_LimitText* message to the combo box to limit text to *TextLen* characters.

ShowList

```
procedure ShowList;
```

Forces the display of the drop down list of a drop down or drop down list combo box.

Store

```
procedure Store(var S: TStream);
```

Stores the combo box on the stream *S* by first calling *TListBox.Store* and then writing the additional fields (*Style, TextLen*) introduced by *TComboBox*.

See also: *TListBox.Store*

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word); virtual;
```

Transfers data to or from the record pointed to by *DataPtr*. The record should be a pointer to a string collection that holds the entries of the combo box's list, then an array of characters holding the currently selected entry. The transfer buffer looks something like this:

```
type
  TComboXferRec = record
    Strings: PStrCollection;
    Selection: array[0..TextLen-1] of Char;
  end;
```

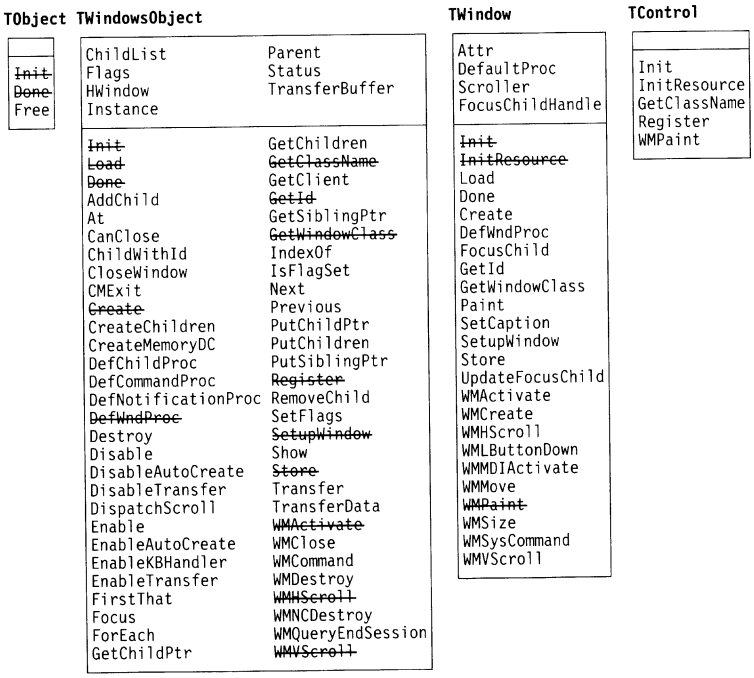
where *TextLen* would be replaced by the value passed in the *Init* constructor.

If *TransferFlag* is *tf_GetData*, the combo box's data is transferred to the *DataPtr* record. If *TransferFlag* is *tf_SetData*, the data is transferred to the combo box from the record. In either of these cases, *Transfer* returns the size of the data transferred.

If *TransferFlag* is *tf_SizeData*, *Transfer* returns the size of the transfer data.

See also: *TListBox.Transfer*

TControl



TControl is an abstract object type that, as an ancestor type, unifies all of the control object types, such as *TScrollBar* and *TButton*. It is also ancestor to *TMDIClient*, a specialized control for MDI-compliant applications.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnId: Integer; ATitle:
  PChar; X, Y, W, H: Integer);
```

Constructs a control object with the passed parent window (*AParent*), control ID (*AnId*), associated text (*ATitle*), position relative to the origin of the parent window's client area (*X*, *Y*), width (*W*), and height (*H*). With these arguments, it fills the control's *Attr* field inherited from *TWindow*. As a default, it sets *Attr.Style* to *ws_Child* **or** *ws_Visible* **or** *ws_Group* **or** *ws_TabStop*, so that all control objects are visible child windows.

InitResource

```
constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
```

Associates a control object with the control element in the resource specified by *ResourceID*. Calls *TWindow.InitResource* and *EnableTransfer* to enable the transfer mechanism.

See also: *TWindow.InitResource, TWindowsObject.EnableTransfer*

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Always

Abstract method to be overridden by descendant objects.

Register

```
function Register: Boolean; virtual;
```

Override: Never

Simply returns *True* to indicate that *TControl* descendants use preregistered window classes.

WMPaint

```
procedure WMPaint(var Msg: TMessage); virtual wm_First + wm_Paint;
```

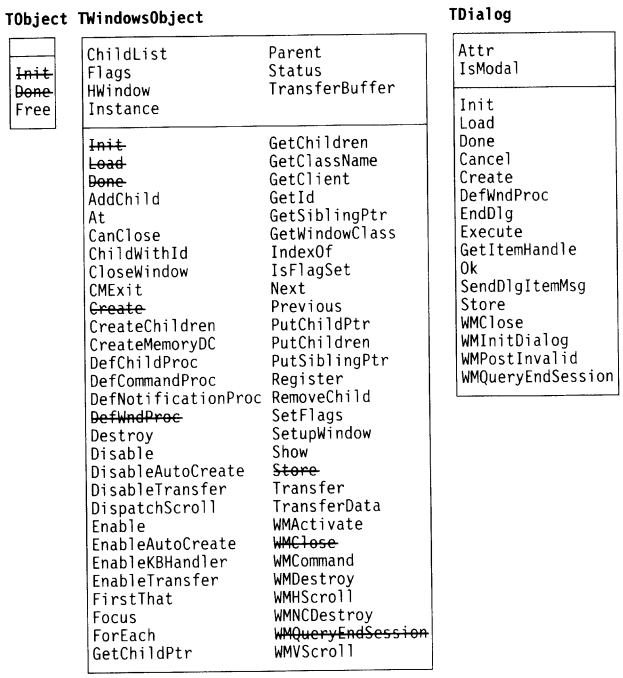
Override: Seldom

Calls *DefWndProc* for standard repainting of control objects.

See also: *TControl.DefWndProc*

TDialog

ODialogs



TDialog defines objects that serve as the modal and modeless dialog boxes used in Windows applications. A *TDialog* object has an associated dialog resource that describes the appearance and placement of its controls. This resource is specified in the call to *TDialog.Init*.

Dialog box objects can be associated with either modal or modeless dialog elements, through calls to its *Execute* or *Create* methods, respectively. Normally, however, dialog objects should be activated through the *TApplication* methods *ExecDialog* and *MakeWindow*, which check for low memory conditions before calling *Execute* or *Create*.

Note that the creation of a modal dialog disables the continued operations of its parent window.

Fields

Attr

```
Attr: TDialogAttr;
```

Attr holds the dialog box's creation attributes in a record of type *TDialogAttr*. The *Attr.Name* field holds the name or ID of the dialog

resource. The *Attr.Param* field holds a parameter that is passed to the dialog procedure when the dialog is created.

IsModal

```
IsModal: Boolean;
```

Read only

IsModal is *True* if the dialog is modal and *False* if it is modeless.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AName: PChar);
```

Override: Sometimes

Constructs the dialog box object by calling the *Init* constructor inherited from *TWindowsObject*, passing the parent window, *AParent*. Sets *Attr.Name* to the string passed in *AName*. The string can be the symbolic name of the dialog resource, such as 'EMPLOYEEINFO' or an integer ID cast to type *PChar*, such as *MakeIntResource(120)*. *Init* also calls *DisableAutoCreate* so that dialog boxes are not automatically created and displayed along with their parent windows.

See also: *TWindowsObject.Init, TWindowsObject.DisableAutoCreate*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a dialog box from the stream *S* by first calling *TWindowsObject.Load* and then reading the additional fields (*Attr* and *IsModal*) introduced by *TDialog*.

See also: *TWindowsObject.Load*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Disposes of the dialog box object by disposing of the string assigned to *Attr.Name*, then calling the *Done* destructor inherited from *TWindowsObject*.

See also: *TWindowsObject.Done*

Cancel

```
procedure Cancel(var Msg: TMessage); virtual id_First + id_Cancel;
```

Override: Sometimes

Automatically responds to the pressing of a dialog's Cancel button by calling *EndDlg* with the value *id_Cancel*.

See also: *TDialog.EndDlg*

Create

```
function Create: Boolean; virtual;
```

Override: Never

Creates a modeless dialog object's corresponding dialog element. *TDialog.Create* returns *True* if successful. If unsuccessful, it calls *Error* with error code *em_InvalidWindow*.

See also: *TDialog.Execute, TWindowsObject.Error*

DefWndProc

```
procedure DefWndProc(var Msg: TMessage); virtual;
```

Override: Never

Elicits Windows' default message processing by setting the result of the passed message equal to zero.

EndDlg

```
procedure EndDlg(ARetValue: Integer); virtual;
```

Override: Never

Destroys modal and modeless dialog boxes. *ARetValue* is passed back as the return value from *TDialog.Execute* for modal dialog boxes.

See also: *TDialog.Execute, TDialog.Create*

Execute

```
function Execute: Integer; virtual;
```

Override: Never

Creates and displays a modal dialog object's corresponding dialog element. This method executes during the entire time the dialog box is on screen, until its *EndDlg* method is called. Before the method finishes, it resets *HWindow* to 0. *TDialog.Execute* returns the integer value returned by *TDialog.EndDlg* if successful. If unsuccessful, *Execute* calls *Error* with the error code *em_InvalidWindow*.

See also: *TDialog.EndDlg, TDialog.Create, TWindowsObject.Error*

GetItemHandle

```
function GetItemHandle(DlgItemID: Integer): HWnd;
```

Override: Never

Returns the handle to the dialog's control identified by the passed ID, *DlgItemID*.

Ok

```
procedure Ok(var Msg: TMessage); virtual id_First + id_OK;
```

Override: Sometimes

Automatically responds to the pressing of a dialog's OK button by calling *CanClose*, and *EndDlg* with the value *id_OK*. Also calls *TransferData(tf_GetData)* to transfer data from the controls to the transfer buffer.

See also: *TDialog.EndDlg*

SendDlgItemMsg

```
function SendDlgItemMsg(DlgItemID: Integer; AMsg, WParam: Word; LParam: Longint):
  Longint;
```

Override: Never

Sends a Windows control message, identified by *AMsg*, to the dialog's control identified by its passed ID, *DlgItemID*. *WParam* and *LParam* become parameters in the Windows message. *SendDlgItemMsg* returns the value returned by the control, or zero if the control ID is invalid.

Store

```
procedure Store(var S: TStream);
```

Stores the dialog box on the stream *S* by first calling *TWindowsObject.Store* and then writing the additional fields (*Attr* and *IsModal*) introduced by *TDialog*.

See also: *TWindowsObject.Store*

WMClose

```
procedure WMClose(var Msg: TMessage); virtual wm_First + wm_Close;
```

If the dialog box is modal, calls *EndDlg(id_Cancel)* to close the dialog box. If the dialog box is modeless, it calls its *WMClose* method inherited from *TWindowsObject*.

See also: *TDialog.EndDlg, TWindowsObject.WMClose*

WMInitDialog

```
procedure WMInitDialog(var Msg: TMessage); virtual wm_First + wm_InitDialog;
```

Override: Never

TDialog.WMInitDialog is automatically called just before the dialog is displayed. It calls *SetupWindow* to perform any initialization needed for the dialog or its controls.

See also: *TWindowsObject.SetupWindow*

WMPostInvalid

```
procedure WMPostInvalid(var Msg: TMessage);
  virtual wm_First + wm_PostInvalid;
```

Responds to a message from a child edit control indicating that its contents are invalid by returning the input focus to that edit control and calling the *Error* method of the edit control's validator.

WMQueryEndSession

```
procedure WMQueryEndSession(var Msg: TMessage);
  virtual wm_First + wm_QueryEndSession;
```

Responds to a message from Windows that Windows is trying to terminate by calling *CanClose*, negating the result, and setting *Msg.Result* to the negated result. Note that Windows expects dialog boxes to return a result *opposite* that returned by other windows. If the dialog box is the application's main window, it calls *Application^.CanClose* instead of its own *CanClose*.

TDialogAttr type

ODialogs

T

Declaration

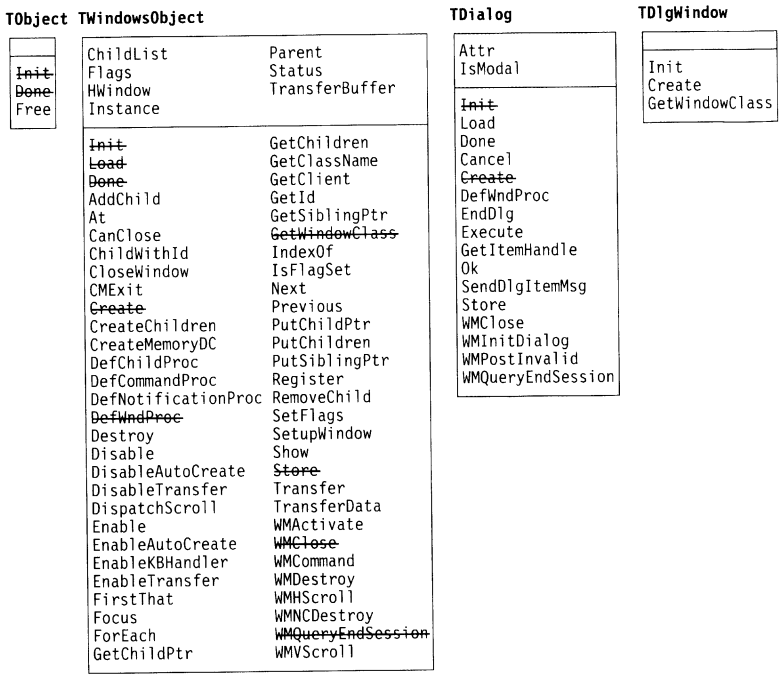
```
TDialogAttr = record
  Name: PChar;
  Param: Longint;
end;
```

Function *TDialog* objects store their attribute values in a record of type *TDialogAttr*.

See also *TDialog.Attr*

TDlgWindow

ODialogs



Dialog windows, defined by *TDlgWindow*, combine some characteristics of dialogs and of windows. Like a dialog box, a dialog window has an associated dialog resource which describes the appearance and position of its controls. However, like a window, it has a window class that can specify icons and cursors. To create and display dialog windows, use the modeless *MakeWindow* method. Do not use the *ExecDialog* method.

The class name of the dialog window's resource (as defined in the Resource Compiler script or dialog editor) must match the class name of the *TDlgWindow* object instance. If the class names do not match, the one given in the resource template will be used.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AName: PChar);
```

Constructs a new *TDlgWindow* object by calling *TDialog.Init*. It also calls *EnableAutoCreate* so that, as a child window, it is automatically created and displayed along with its parent window.

See also: *TDialog.Init*, *TWindowsObject.EnableAutoCreate*

Create `function Create: Boolean; virtual;`

Override: Never

Registers the dialog window's class and calls *TDialog.Create*. *TDlgWindow.Create* returns *True* if successful.

See also: *TWindowsObject.Register, TDialog.Create*

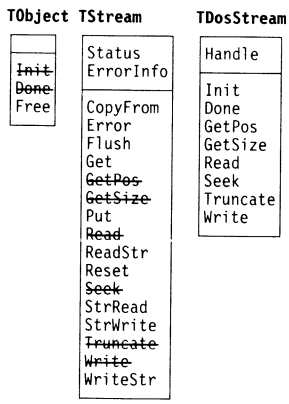
GetWindowClass `procedure GetWindowClass(var AWndClass: TWndClass); virtual;`

Override: Often

Defines the default window class record and passes it back in *AWndClass*. This window class specifies no menu and a standard icon and cursor. Redefine *GetWindowClass*, as well as *GetClassName* for your *TDlgWindow* descendants. However, make sure your *GetWindowClass* method calls *TDlgWindow.GetWindowClass* before modifying any *TWndClass* fields.

TDosStream

Objects



TDosStream is a specialized *TStream* derivative implementing unbuffered DOS file streams. The constructor lets you create or open a DOS file by specifying its name and access mode: *stCreate, stOpenRead, stOpenWrite,* or *stOpen*. The one additional field of *TDosStream* is *Handle*, the traditional DOS file handle used to access an open file. Most applications will use the buffered derivative of *TDosStream* called *TBufStream*. *TDosStream* overrides all the abstract methods of *TStream* except for *TStream.Flush*.

Fields

Handle `Handle: Word;` **Read only**

Handle is the DOS file handle used to access an open file stream.

Methods

Init

```
constructor Init(FileName: FNameStr; Mode: Word);
```

Creates a DOS file stream with the given file name and access mode. If successful, the *Handle* field is set with the DOS file handle. Failure is signaled by a call to *Error* with an argument of *stInitError*.

The *Mode* argument must be set to one of the values *stCreate*, *stOpenRead*, *stOpenWrite*, or *stOpen*. These constant values are explained in this chapter under "*stXXXX* stream constants."

Done

```
destructor Done; virtual;
```

Override: Never

Closes and disposes of the DOS file stream.

See also: *TDosStream.Init*

GetPos

```
function GetPos: Longint; virtual;
```

Override: Never

Returns the value of the stream's current position.

See also: *TDosStream.Seek*

GetSize

```
function GetSize: Longint; virtual;
```

Override: Never

Returns the total size in bytes of the stream.

Read

```
procedure Read(var Buf; Count: Word); virtual;
```

Override: Never

Reads *Count* bytes into the *Buf* buffer starting at the stream's current position.

See also: *TDosStream.Write*, *stReadError*

Seek

```
procedure Seek(Pos: Longint); virtual;
```

Override: Never

Resets the current position to *Pos* bytes from the beginning of the stream.

See also: *TDosStream.GetPos*, *TDosStream.GetSize*

Truncate

```
procedure Truncate; virtual;
```

Override: Never

Deletes all data on the stream from the current position to the end.

See also: *TDosStream.GetPos*, *TDosStream.Seek*

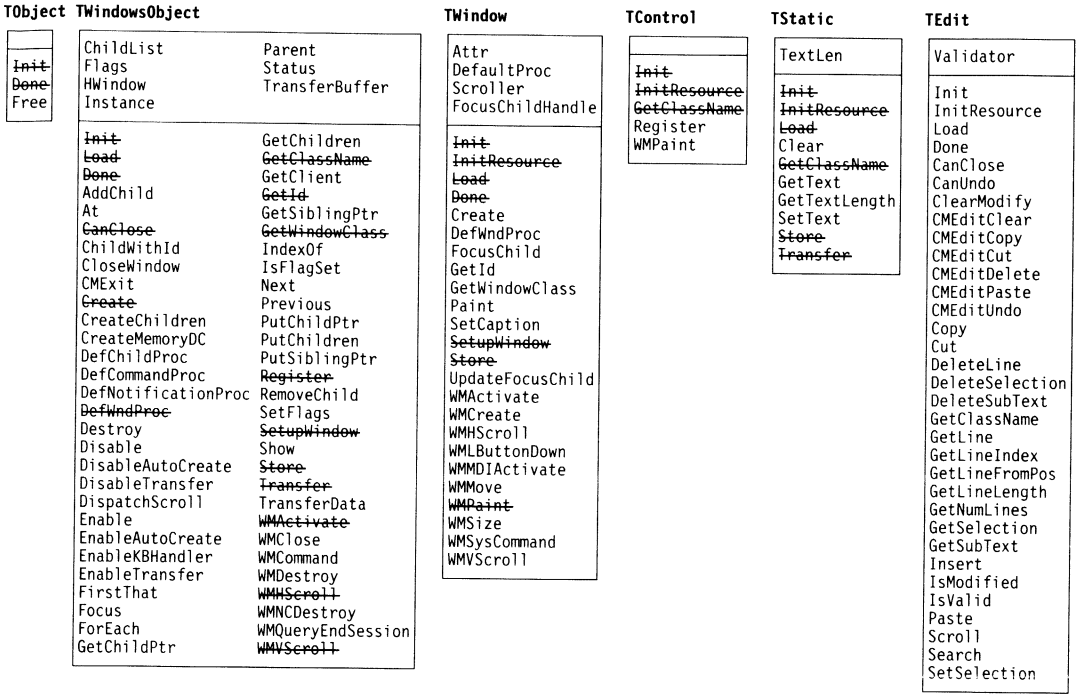
Write

```
procedure Write(var Buf; Count: Word); virtual;
```

Writes *Count* bytes from the *Buf* buffer to the stream, starting at the current position.

See also: *TDosStream.Read*, *stWriteError*

TEdit



TEdit is an interface object that represents a corresponding edit control element in Windows.

There are two styles of edit control objects: single line and multiline. Multiline edit controls allow vertical scroll bars and editing in multiple lines. Most of *TEdit's* methods manage the edit control's text. *TEdit* also includes some command-based message response methods for automatically responding to cut, copy, paste, delete, clear and undo menu selections from the edit control's parent window. Two important methods inherited from *TEdit's* ancestor, *TStatic*, are *GetText* and *SetText*.

In this version of ObjectWindows, edit controls support data validation through the use of validator objects. Data validation is described in Chapter 13, "Data validation."

Field

Validator

```
Validator: PValidator;
```

Points to the edit control's associated validator. If the edit control has no validator, *Validator* is **nil**.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnId: Integer; ATitle:
  PChar; X, Y, W, H, ATextLen: Integer; Multiline: Boolean);
```

Override: Sometimes

Constructs an edit control object with a parent window (*AParent*), and fills its *Attr* fields with the passed control ID (*AnId*), initial text (*ATitle*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), height (*H*), and text buffer length (*ATextLen*).

If *ATextLen* is zero, there is no explicit limit on the number of characters that can be entered. If *Multiline* is *True*, the edit control will be a multiline edit control with horizontal and vertical scroll bars. In that case, the *Attr.Style* field will include the Windows style constants *es_Multiline*, *es_AutoVScroll*, *es_AutoHScroll*, *es_Left*, *ws_VScroll*, and *ws_HScroll*. If *Multiline* is *False*, the edit control will have a single line of text and will have a border (*ws_Border*) and will be left justified (*es_Left*).

InitResource

```
constructor InitResource(AParent: PWindowsObject; ResourceID: Word;
  ATextLen: Word);
```

Constructs an edit control object and associates it with the screen element in the resource specified by *ResourceID* by calling the *InitResource* constructor inherited from *TStatic*. Sets *Validator* to **nil**.

See also: *TStatic.InitResource*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads an edit control from the stream *S* by first calling the *Load* constructor inherited from *TStatic*, then reading the additional field (*Validator*) introduced by *TEdit*.

See also: *TStatic.Load*

Done

```
destructor Done; virtual;
```

Disposes of the edit control's associated validator object by calling *SetValidator* with a parameter of **nil**, then disposes of the edit control object by calling the *Done* destructor inherited from *TStatic*.

See also: *TEdit.SetValidator, TStatic.Done*

CanClose

```
function CanClose: Boolean;
```

Calls the *CanClose* method inherited from *TStatic*, and if that returns *False*, *CanClose* also returns *False*. If the inherited *CanClose* returns *True*, *CanClose* then calls *IsValid(True)*. If *IsValid* returns *True*, *CanClose* also returns *True*; otherwise, it returns *False*, and sets the input focus to the edit control.

See also: *TEdit.IsValid, TStatic.CanClose*

CanUndo

```
function CanUndo: Boolean; virtual;
```

Override: Seldom

Returns *True* if it is possible to undo the last edit.

See also: *TEdit.Undo*

ClearModify

```
procedure ClearModify; virtual;
```

Override: Seldom

Resets the changes flag for the edit control.

See also: *TEdit.IsModified*

CMEditClear

```
procedure CMEditClear(var Msg: TMessage); virtual cm_First + cm_EditClear;
```

Override: Never

Automatically responds to a menu selection with a menu ID of *cm_EditClear* by calling the *Clear* method.

See also: *TEdit.Clear*

CMEditCopy

```
procedure CMEditCopy(var Msg: TMessage); virtual cm_First + cm_EditCopy;
```

Override: Never

Automatically responds to a menu selection with a menu ID of *cm_EditCopy* by calling the *Copy* method.

See also: *TEdit.Copy*

CMEditCut

```
procedure CMEditCut(var Msg: TMessage); virtual cm_First + cm_EditCut;
```

Override: Never

Automatically responds to a menu selection with a menu ID of *cm_EditCut* by calling the *Cut* method.

See also: *TEdit.Cut*

CMEditDelete

```
procedure CMEditDelete(var Msg: TMessage); virtual cm_First + cm_EditDelete;
```

Override: Never

Automatically responds to a menu selection with a menu ID of *cm_EditDelete* by calling the *DeleteSelection* method.

See also: *TEdit.DeleteSelection*

CMEditPaste

```
procedure CMEditPaste(var Msg: TMessage); virtual cm_First + cm_EditPaste;
```

Override: Never

Automatically responds to a menu selection with a menu ID of *cm_EditPaste* by calling the *Paste* method.

See also: *TEdit.Paste*

CMEditUndo

```
procedure CMEditUndo(var Msg: TMessage); virtual cm_First + cm_EditUndo;
```

Override: Never

Automatically responds to a menu selection with a menu ID of *cm_EditUndo* by calling the *Undo* method.

See also: *TEdit.Undo*

Copy

```
procedure Copy; virtual;
```

Override: Seldom

Copies the currently selected text into the clipboard.

See also: *TEdit.CMEditCopy*

Cut

```
procedure Cut; virtual;
```

Override: Seldom

Copies the currently selected text into the clipboard and deletes it from the edit control.

See also: *TEdit.CMEditCut*

DeleteLine

```
function DeleteLine(LineNumber: Integer): Boolean; virtual;
```

Override: Sometimes

Deletes the text in the line specified by *LineNumber* in a multiline edit control. *DeleteLine* does not delete the line break and affects no other lines. The method returns *True* if successful.

DeleteSelection

```
function DeleteSelection: Boolean; virtual;
```

Override: Seldom

Clears the currently selected text, and returns *False* if no text is selected.

See also: *TEdit.CMEditDelete*

DeleteSubText

```
function DeleteSubText(StartPos, EndPos: Integer): Boolean; virtual;
```

Override: Seldom

Deletes the text between the starting and ending positions specified by *StartPos* and *EndPos*. The first character is in position zero, and the position numbers continue sequentially throughout all lines in a multiline edit control. Line breaks count as two characters. *DeleteSubText* returns *True* if successful.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Never

Returns the name of *TEdit's* window class, 'Edit'.

GetLine

```
function GetLine(ATextString: PChar; StrSize, LineNumber: Integer): Boolean;
  virtual;
```

Override: Seldom

Retrieves a multiline edit control's text from the line specified by *LineNumber* and returns it in *ATextString*. *StrSize* indicates how many characters to retrieve. *GetLine* returns *False* if it is unable to retrieve the text or if it is too long.

See also: *TStatic.GetText*, *TEdit.GetNumLines*, *TEdit.LineLength*

GetLineFromPos

```
function GetLineFromPos(CharPos: Integer): Integer; virtual;
```

Override: Seldom

Returns, from a multiline edit control, the line number on which the character position specified by *CharPos* occurs. The position of the first character is zero and the numbers continue sequentially throughout all of the lines. Line breaks count as two characters.

GetLineIndex

```
function GetLineIndex(LineNumber: Integer): Integer; virtual;
```

Override: Seldom

Returns, from a multiline edit control, the number of characters that appear before the line number specified by *LineNumber*. Line breaks count as two characters. If the specified line does not exist, *GetLineIndex* returns the total number of characters in the edit control.

GetLineLength

```
function GetLineLength(LineNumber: Integer): Integer; virtual;
```

Override: Seldom

Returns, from a multiline edit control, the number of characters in the line specified by *LineNumber*. *GetLineLength* should be called before calling *GetLine*.

See also: *TEdit.GetLine*

GetNumLines

```
function GetNumLines: Integer; virtual;
```

Override: Seldom

Returns the number of lines that have been entered in a multiline edit control. *GetNumLines* should be called before calling *GetLine*.

See also: *TEdit.GetLine*

GetSelection

```
procedure GetSelection(var StartPos, EndPos: Integer); virtual;
```

Override: Seldom

Retrieves the starting and ending positions of the currently selected text and returns them in the *StartPos* and *EndPos* arguments. The first character is at position zero. In a multiline edit control, the positions are counted sequentially throughout all lines, and line breaks count for two characters. When using *GetSelection* in conjunction with *GetSubText*, you can get the currently selected text.

See also: *TEdit.GetSubText*

GetSubText

```
procedure GetSubText(ATextString: PChar; StartPos, EndPos: Integer);
  virtual;
```

Override: Seldom

Retrieves, in *ATextString*, the text in an edit control from indexes *StartPos* to *EndPos*. When specifying the range, the first character is at index zero. In a multiline edit control, the characters are counted sequentially throughout all of the lines, and line breaks are counted as two characters.

See also: *TEdit.GetSelection*

Insert

```
procedure Insert(ATextString: PChar); virtual;
```

Override: Seldom

Inserts the text passed in *ATextString* into the edit control at the current text insertion point and replaces any currently selected text. *Insert* is similar to *Paste* but does not affect the clipboard.

See also: *TEdit.Paste*

IsModified

```
function IsModified: Boolean; virtual;
```

Override: Seldom

Returns *True* if the user has changed the text in the edit control.

See also: *TEdit.ClearModify*

IsValid

```
function IsValid(ReportError: Boolean): Boolean;
```

Returns *True* if the edit control's text is valid. *IsValid* always returns *True* if the text has more than one line or if the control has no associated validator object. If the control has a validator and only a single line of text, *IsValid* calls the validator's *Valid* method if *ReportError* is *True*, or its *IsValid* if *ReportError* is *False*, and returns the value returned by the validator method.

See also: *TValidator.IsValid*, *TValidator.Valid*

Paste

```
procedure Paste; virtual;
```

Override: Seldom

Inserts text from the clipboard into the edit control at the current insertion point.

See also: *TEdit.CMEditPaste*

Scroll

```
procedure Scroll(HorizontalUnit, VerticalUnit: Integer); virtual;
```

Override: Seldom

Scrolls a multiline edit control horizontally and vertically by the numbers of characters specified in *HorizontalUnit* and *VerticalUnit*. Positive values result in a scroll to the right or down, and negative values scroll to the left or up.

Search

```
function Search(StartPos: Integer; AText: PChar;
  CaseSensitive: Boolean): Integer;
```

Search looks through the edit control's text, starting with the character at *StartPos*, until it finds a match for *AText*. If the text is found, the matching

text will be selected, and *Search* returns the position of the start of the matched text. If *AText* is not found in the edit control's text, *Search* returns –1.

Passing –1 in *StartPos* causes the search to begin at the current position.

SetSelection

```
function SetSelection(StartPos, EndPos: Integer): Boolean; virtual;
```

Override: Seldom

Forces the selection of the text between the positions specified by *StartPos* and *EndPos*, but not including the character at *EndPos*. The first character is in position zero, and the position numbers continue sequentially throughout all lines in a multiline edit control. Line breaks count as two characters.

SetupWindow

```
procedure SetupWindow; virtual;
```

Sets up the edit control by calling the *SetupWindow* method inherited from *TStatic*. If the *TextLen* field is nonzero, limits the number of characters the user can type into the edit control by sending an *em_LimitText* message to the screen element.

See also: *TStatic.SetupWindow, em_LimitText* message

SetValidator

```
procedure SetValidator(AValid: PValidator);
```

Disposes of any existing validator, then sets *Validator* to *AValid*.

Store

```
procedure Store(var S: TStream);
```

Stores the edit control on the stream *S* by first calling *TStatic.Store* and then writing the additional field (*IsMultiline*) introduced by *TEdit*.

See also: *TStatic.Store*

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word;
  virtual;
```

Override: Sometimes

Transfers *TextLen* characters of the current text of the edit control to or from the memory location pointed to by *DataPtr*. If *TransferFlag* is *tf_GetData*, the text is transferred to the memory location. If *TransferFlag* is *tf_SetData*, the edit control's text is set to the text at the memory location. *Transfer* returns *TextLen*, the number of bytes stored in or retrieved from the memory location. If *TransferFlag* is *tf_SizeData*, *Transfer* returns the size of the transfer data.

Undo

```
procedure Undo; virtual;
```

Override: Seldom

Undoes the last edit.

See also: *TEdit.CanUndo, TEdit.CMEditUndo*

WMChar

```
procedure WMChar(var Msg: TMessage); virtual wm_First + wm_Char;
```

Processes incoming characters by calling *DefWndProc*. If the edit control has a validator and only a single line of text, *WMChar* passes the current text to the validator's *IsValidInput* method. If *IsValidInput* returns *False*, *WMChar* restores the edit control's text to its state before inserting the current character.

See also: *TValidator.IsValidInput*

WMGetDlgCode

```
procedure WMGetDlgCode(var Msg: TMessage); virtual wm_First + wm_GetDlgCode;
```

WMGetDlgCode controls whether the edit control allows the user to move the input focus out of the control by typing *Tab*. *WMGetDlgCode* calls *DefWndProc* to handle the default message processing, then calls *IsValid* to validate the edit control text. If *IsValid* returns *False*, *WMGetDlgCode* sets *Msg.Result* to *Msg.Result* **or** *dlgc_WantTab*, which causes the edit control to take over processing of *Tab*, instead of letting Windows handle the change in focus as it usually would.

See also: *TEdit.IsValid*

WMKeyDown

```
procedure WMKeyDown(var Msg: TMessage); virtual wm_First + wm_KeyDown;
```

WMKeyDown traps *Tab* keystrokes if the edit control's contents are invalid. Normally, Windows handles *Tab* in dialog boxes by moving the input focus, without notifying the control losing focus. However, by using *WMGetDlgCode*, an edit control with invalid contents can force Windows to pass it *Tab*s.

If *WMKeyDown* detects a *Tab*, it calls *IsValid*, and if *IsValid* returns *False*, *WMKeyDown* bypasses the normal processing of the keystroke. That is, the edit control won't let Windows move the input focus if the contents of the edit control are invalid.

See also: *TEdit.IsValid*, *TEdit.WMGetDlgCode*

WMKillFocus

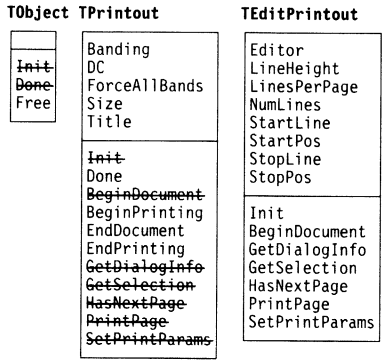
```
procedure WMKillFocus(var Msg: TMessage); virtual wm_First + wm_KillFocus;
```

Windows sends a *wm_KillFocus* message to the control when it wants to move the input focus from the control. *WMKillFocus* validates the edit control contents unless the focus is being taken by another application, a Cancel button, or an OK button. If the edit control's contents are invalid, *WMKillFocus* posts a *wm_PostInvalid* message to its parent dialog box, which responds by returning the focus to the edit control.

See also: *TDialog.WMPostInvalid*

TEditPrintout

OPrinter



TEditPrintout is a printout object designed to print the contents of an edit control.

Fields

Editor

```
Editor: PEdit;
```

Points to the edit control to print.

LineHeight

```
LineHeight: Integer;
```

Height of a printed line, calculated by *SetPrintParams* based on the text metrics of the printout device.

LinesPerPage

```
LinesPerPage: Integer;
```

Number of text lines to print per page, calculated by *SetPrintParams* by dividing the size of the device context by *LineHeight*.

NumLines

```
NumLines: Integer;
```

Set by *SetPrintParams* to the number of text lines in the edit control.

StartLine

```
StartLine: Integer;
```

BeginDocument sets *StartLine* to zero, meaning the first line in the text. If the user chooses to print selected text, *GetSelection* resets *StartLine* to the number of the line containing the first selected character.

StartPos

```
StartPos: Integer;
```

If printing selected text, indicates the position of the first selected character in the edit control's text; otherwise, zero.

StopLine

```
StopLine: Integer;
```

BeginDocument sets *StopLine* to *NumLines* – 1, meaning the last line in the text. If the user chooses to print selected text, *GetSelection* resets *StopLine* to the number of the line containing the last selected character.

StopPos

```
StopPos: Integer;
```

If printing selected text, indicates the position of the last selected character in the control's text; otherwise, 32767.

Methods

Init

```
constructor Init(AEditor: PEdit; ATitle: PChar);
```

Constructs an edit printout object by first calling the *Init* constructor inherited from *TPrintout*, then setting *Editor* to *AEditor* and initializing all other fields to zero.

See also: *TPrintout.Init*

BeginDocument

```
procedure BeginDocument(StartPage, EndPage: Integer; Flags: Word); virtual;
```

Checks the *pf_Selection* bit in the *Flags* field to determine whether the user wants to print selected text or the entire edit text. If only the selection, *BeginDocument* does nothing, leaving the values set by *GetSelection*. Otherwise, sets *StartLine* and *StartPos* to zero, *StopLine* to *NumLines* – 1, and *StopPos* to 32767.

GetDialogInfo

```
function GetDialogInfo(var Pages: Integer): Boolean; virtual;
```

Sets *Pages* to *NumLines* **div** *LinesPerPage* + 1 and returns *True*, indicating that the specific number of pages should print, regardless of the value returned from *HasNextPage*.

GetSelection

```
function GetSelection(var Start, Stop: Integer): Boolean; virtual;
```

Sets *StartPos* and *StopPos* to the beginning and ending positions of the selected text by calling *Editor^.GetSelection*. If *StartPos* and *StopPos* are equal, meaning no text selected, *GetSelection* returns *False*. Otherwise, *GetSelection* sets *StartLine* and *StopLine* to the numbers of the lines containing *StartPos* and *StopPos*, respectively, then sets *Start* to one and *Stop* to the number of pages needed to print the selected text lines, then returns *True*.

HasNextPage

```
function HasNextPage: Boolean; virtual;
```

Returns *True* always. Because the edit printout object can calculate exactly how many pages it takes to print the edit control's text, it does not rely on *HasNextPage* to indicate that printing has finished.

PrintPage

```
procedure PrintPage(Page: Word; var Rect: TRect; Flags: Word); virtual;
```

For each line on the page, *PrintPage* takes a line of text from *Editor* and sends it to the device context using *TextOut*. If the current line is either the first or last line of selected text, *PrintPage* strips off the unselected portions of the line before calling *TextOut*.

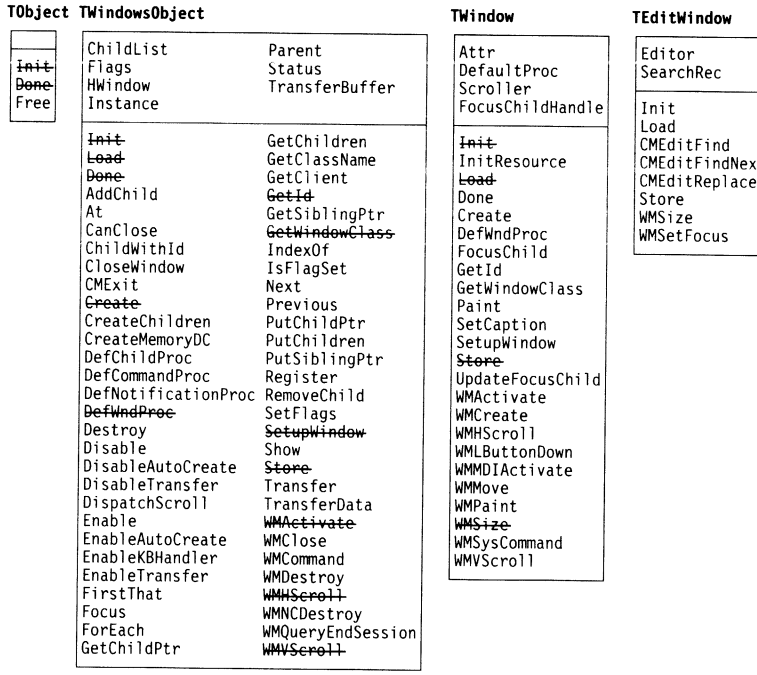
SetPrintParams

```
procedure SetPrintParams(ADC: HDC; ASize: TPoint); virtual;
```

Sets the edit printout object's device context and print area size by calling the *SetPrintParams* method inherited from *TPrintout*, then gets the number of lines in *Editor* by calling its *GetNumLines* method. Calculates *LineHeight* and *LinesPerPage* based on the text metrics of the device context passed in *ADC*.

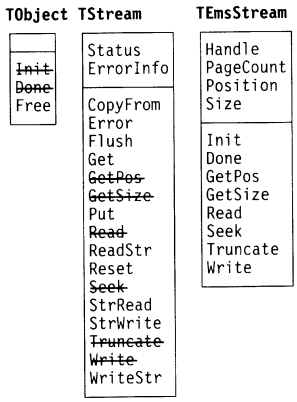
See also: *TPrintout.SetPrintParams*

TEditWindow — OStdWnds



An edit window is a window whose entire client area is filled by an edit control. For details on the fields and methods of the type *TEditWindow*, see the online Help.

TEmsStream Objects



TEmsStream is a specialized *TStream* derivative for implementing streams in EMS memory. The additional fields provide an EMS handle, page count, stream size, and current position. *TEmsStream* overrides the six abstract methods of *TStream* as well as providing a specialized constructor and destructor.

☞ When debugging a program using EMS streams, the IDE cannot recover EMS memory allocated by your program if your program terminates prematurely or if you forget to call the *Done* destructor for an EMS stream. Only the *Done* method (or rebooting) can release the EMS pages owned by the stream.

Fields

Handle `Handle: Word;` **Read only**

The EMS handle for the stream.

PageCount `PageCount: Word;` **Read only**

The number of allocated pages for the stream, with 16K per page.

Position `Position: Longint;` **Read only**

The current position within the stream. The first position is 0.

Size `Size: Longint;` **Read only**

The size of the stream in bytes.

Methods

Init `constructor Init(MinSize, MaxSize: Longint);`

Creates an EMS stream with the given minimum and maximum sizes in bytes. Calls *TStream.Init*, then sets *Handle*, *Size* and *PageCount*. Calls *Error* with an argument of *stInitError* if initialization fails.

See also: *TEmsStream.Done*

Done `destructor Done; virtual;`

Override: Never

Disposes of the EMS stream and releases EMS pages used.

See also: *TEmsStream.Init*

GetPos `function GetPos: Longint; virtual;`

Override: Never

Returns the value of the stream's current position.

See also: *TEmsStream.Seek*

GetSize `function GetSize: Longint; virtual;`

Override: Never

Returns the total size of the stream.

Read `procedure Read(var Buf; Count: Word); virtual;`

Override: Never

Reads *Count* bytes into the *Buf* buffer starting at the stream's current position.

See also: *TEmsStream.Write*, *stReadError*

Seek `procedure Seek(Pos: Longint); virtual;`

Override: Never

Resets the current position to *Pos* bytes from the start of the stream.

See also: *TEmsStream.GetPos*, *TEmsStream.GetSize*

Truncate `procedure Truncate; virtual;`

Override: Never

Deletes all data on the stream from the current position to the end. The current position is set to the new end of the stream.

See also: *TEmsStream.GetPos*, *TEmsStream.Seek*

Write `procedure Write(var Buf; Count: Word); virtual;`

Override: Never Writes *Count* bytes from the *Buf* buffer to the stream, starting at the current position.

See also: *TStream.Read, TEmsStream.GetPos, TEmsStream.Seek*

tf_XXXX constants — OWindows

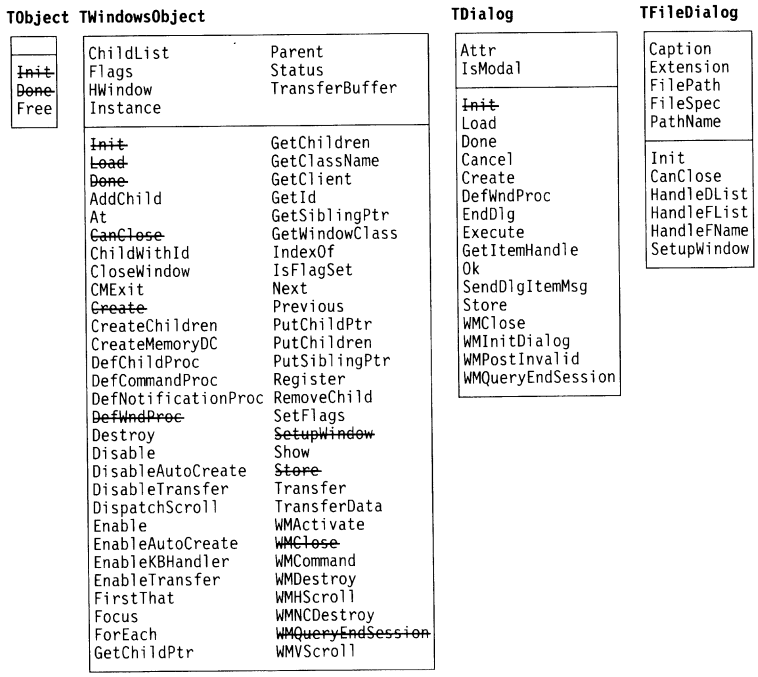
Function The *Transfer* method uses flag constants beginning with *tf_*.

Values The following constants are defined:

Table 21.25 Transfer function constants

Constants	Value	Meaning
tf_SizeData	0	Find out the size of data transferred by the object.
tf_GetData	1	Retrieve data from the object.
tf_SetData	2	Send data to set the value of the object.

TFileDialog — OStdDlgs



File dialog boxes allow the user to choose a file to open, or to name a file to save into.

Fields

Caption

```
Caption: PChar;
```

Points to the string that appears in the title bar of the dialog box. If *Caption* is **nil**, the text from the dialog resource is used. You can customize the title by assigning a different string to *Caption*.

Extension

```
Extension: array[0..fsExtension] of Char;
```

Holds the extension for files. If the user types in a file name with no extension, the file dialog box appends *Extension* to the name.

FilePath

```
FilePath: PChar;
```

Set to the value passed in the constructor, *FilePath* points to a buffer that will hold the path name chosen by the user. The file dialog box puts the final file name into the buffer.

FileSpec

```
FileSpec: array[0..fsFileSpec] of Char;
```

Holds the file specification, usually a file name with wildcard characters, which the file dialog box appends to the currently selected path.

PathName

```
PathName: array[0..fsPathName] of Char;
```

Holds the current selected directory path.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AName, AFilePath: PChar);
```

Constructs a file dialog box with the parent window *AParent* from a dialog resource. The resource is determined by the value passed in *AName*. *AName* should hold one of the *sd_XXXX* constants, *sd_FileOpen* or *sd_FileSave*, cast into a *PChar* with *MakeIntResource*, to determine whether to construct a file open dialog box or a file save dialog box. Based on *AName* and the value in *BWCCClassNames*, *Init* determines what resource to specify to the *Init* constructor inherited form *TDialog*. Sets *Caption* to **nil** and *FilePath* to *AFilePath*. *AFilePath* must point to a buffer large enough to hold a complete path name, usually an **array**[0..fsPathName] **of** *Char*.

CanClose

```
function CanClose: Boolean; virtual;
```

Returns *True* if the user has selected a file, allowing the dialog box to close. If the currently selected path name is a directory or contains wildcard characters, *CanClose* updates the file and directory list boxes and returns *False*, indicating that the dialog box should not close.

SetupWindow

```
procedure SetupWindow; virtual;
```

Limits the number of characters in the file name to *fsPathName*. If *Caption* in non-**nil**, sets the caption of the dialog box to *Caption*. Copies *FilePath* into *PathName* and the extension of *PathName* into *Extension*. Updates the list boxes to match the current path name.

HandleFName

```
procedure HandleFName(var Msg: TMessage); virtual id_First + id_FName;
```

Responds to changes in the *FileName* edit control by enabling or disabling the Ok button. If the file name is empty, *HandleFName* disables Ok; otherwise, it enables Ok.

HandleFList

```
procedure HandleFList(var Msg: TMessage); virtual id_First + id_FList;
```

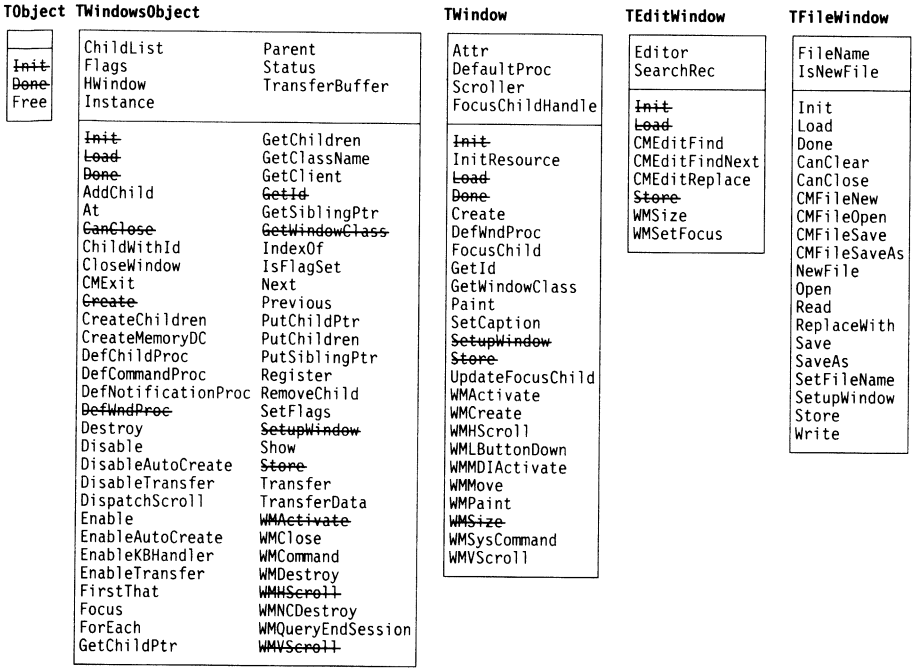
Responds to notification messages from the file list box by updating the current file name and list boxes if the user selected or double-clicked a file. If the user double-clicked, calls *Ok* to terminate the dialog box. If the list box loses the input focus, *HandlFList* unselects whatever item was selected.

HandleDList

```
procedure HandleDList(var Msg: TMessage); virtual id_First + id_DList;
```

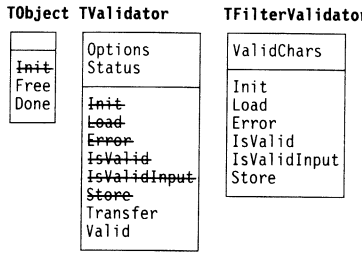
Responds to notification messages from the directory list box by updating the current file name and list boxes if the user selected or double-clicked a directory. If the list box loses the input focus, *HandleDList* unselects whatever item was selected.

TFileWindow — OStdWnds



A file window is an edit window with specialized methods for reading and writing its edit text from a file. For details on the fields and methods of the type *TFileWindow*, see the online Help.

TFilterValidator — Validate



Filter validator objects check an input field as the user types into it. The validator holds a set of allowed characters. If the user types one of the legal characters, the filter validator indicates that the character is valid. If

the user types any other character, the validator indicates that the input is invalid.

Field

ValidChars

```
ValidChars: TCharSet;
```

Contains the set of all characters the user can type. For example, to allow only numeric digits, set *ValidChars* to ['0'..'9']. *ValidChars* is set by the *AValidChars* parameter passed to the *Init* constructor.

Methods

Init

```
constructor Init(AValidChars: TCharSet);
```

Constructs a filter validator object by first calling the *Init* constructor inherited from *TValidator*, then setting *ValidChars* to *AValidChars*.

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a filter validator object from the stream *S* by first calling the *Load* constructor inherited from *TValidator*, then reading the set of valid characters into *ValidChars*.

See also: *TValidator.Load*

Error

```
procedure Error; virtual;
```

Displays a message box indicating that the text string contains an invalid character.

IsValid

```
function IsValid(const S: string): Boolean; virtual;
```

Returns *True* if all characters in *S* are in the set of allowed characters, *ValidChar*; otherwise, returns *False*.

IsValidInput

```
function IsValidInput(var S: string; SuppressFill: Boolean): Boolean;
  virtual;
```

Checks each character in the string *S* to make sure it is in the set of allowed characters, *ValidChar*. Returns *True* if all characters in *S* are valid; otherwise, returns *False*.

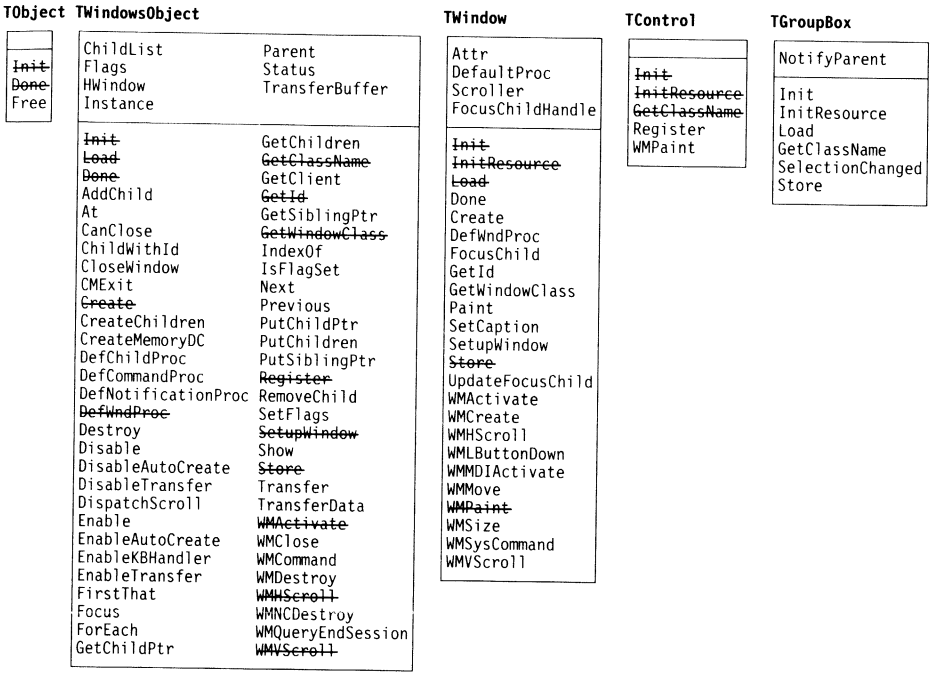
Store

```
procedure Store(var S: TStream);
```

Stores the filter validator object on the stream *S* by writing *ValidChars*.

TGroupBox

ODialogs



TGroupBox is an interface object that represents a corresponding group box element in Windows. While group boxes don't serve an active purpose on the screen, they visually unify a group of selection boxes (check boxes and radio buttons). Behind the scenes, however, they perform the important role in managing the states of their selection boxes. For example, you might want to respond to the user checking one box by unchecking all of the others.

Field

NotifyParent

```
NotifyParent: Boolean;
```

Read/write

Flag that indicates whether the parent window should be notified when the state of the group box's selection boxes has changed.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; AText: PChar;
  X, Y, W, H: Integer);
```

Override: Sometimes

Constructs a group box object with the passed parent window (*AParent*), control ID (*AnId*), associated text (*AText*), position relative to the origin of the parent window's client area (*X*, *Y*), width (*W*), and height (*H*). Calls *TControl.Init* and then adds the Windows style *bs_GroupBox* to, and removes the style *ws_TabStop* from, the group box's *Attr.Style* field. Sets *NotifyParent* to *True*, so by default, the group box's parent is notified when the state of its selection boxes changes.

See also: *TControl.Init*

InitResource

```
constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
```

Associates a group box object with the control element in the resource specified by *ResourceID* by calling the *InitResource* constructor inherited from *TControl*. Calls *DisableTransfer* to exclude group boxes from the transfer mechanism, since they have no data to transfer.

See also: *TControl.InitResource*, *TWindowsObject.DisableTransfer*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a group box from the stream *S* by first calling *TControl.Load* and then reading the additional field (*NotifyParent*) introduced by *TGroupBox*.

See also: *TControl.Load*

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Sometimes

Returns the name of *TGroupBox*'s window class, 'Button'.

SelectionChanged

```
procedure SelectionChanged(ControlId: Integer); virtual;
```

Override: Sometimes

If *NotifyParent* is *True*, notifies the parent window of the group box that one of its selections has changed by sending it a child-ID-based message. This method could be overridden to allow the group box to respond to its selections.

Store

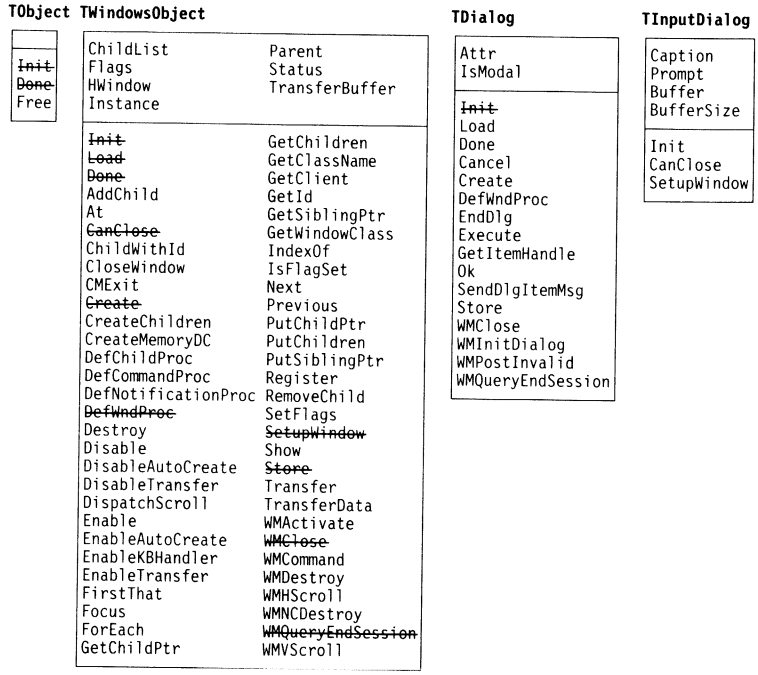
```
procedure Store(var S: TStream);
```

Stores the group box on the stream *S* by first calling *TControl.Store* and then writing the additional field (*NotifyParent*) introduced by *TGroupBox*.

See also: *TControl.Store*

TInputDialog

OStdDlgs



Input dialog boxes are an easy way to prompt the user for a single line of text.

Fields

Buffer `Buffer: PChar;`

Points to a character buffer to hold the user's input. Any characters in *Buffer* at initialization appear as the default text in the input dialog box's edit control.

BufferSize `BufferSize: Word;`

Specifies the size, in bytes, of the buffer pointed to by *Buffer*.

Caption `Caption: PChar;`

Points to a string which appears as the caption of the dialog box.

Prompt

```
Prompt: PChar;
```

Points to the string which appears above the input area.

Methods

Init

```
constructor Init(AParent: PWindowsObject;
  ACaption, APrompt, ABuffer: PChar; ABufferSize: Word);
```

Constructs a file dialog box with the parent window passed in *AParent* by calling the *Init* constructor inherited from *TDialog*. The name of the resource template is determined by looking at *BWCCClassNames*. If the application uses BWCC, the input dialog box will use a BWCC resource. Otherwise, a normal Windows dialog box is used.

Sets *Caption* to *ACaption*, *Prompt* to *APrompt*, *Buffer* to *ABuffer*, and *BufferSize* to *ABufferSize*.

See also: *TDialog.Init*

CanClose

```
function CanClose: Boolean; virtual;
```

Reads up to *BufferSize* characters from the dialog box's edit control into *Buffer*, then returns *True*, indicating that the dialog box can close.

SetupWindow

```
procedure SetupWindow; virtual;
```

Sets up the input dialog box by first calling the *SetupWindow* method inherited from *TDialog*, then setting the dialog box's caption to the string in *Caption*. Sets the prompt text to *Prompt*, the buffer text to *Buffer*, and limits the number of characters in the buffer to *BufferSize* – 1.

TItemList type

Objects

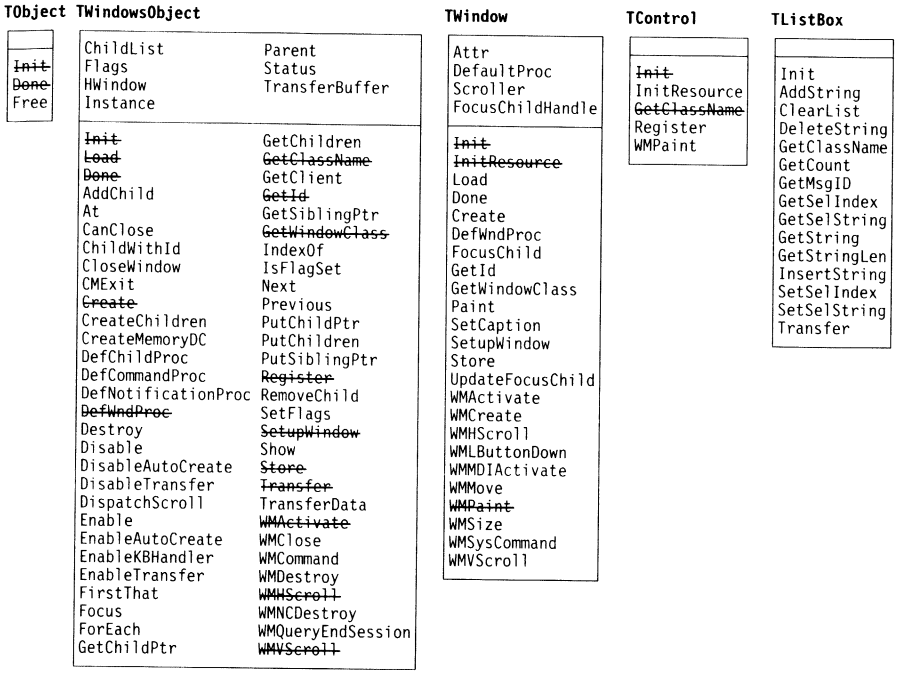
Declaration

```
TItemList = array[0..MaxCollectionSize - 1] of Pointer;
```

Function An array of generic pointers used internally by *TCollection* objects.

TListBox

ODialogs



TListBox is an interface object that represents a corresponding list box element in Windows. *TListBox*'s methods also serve instances of its descendant, *TComboBox*.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnId: Integer; X,Y,W,H: Integer);
```

Override: Sometimes

Constructs a list box object with the passed parent window (*AParent*), control ID (*AnId*), position relative to the origin of the parent window's client area (*X*, *Y*), width (*W*), and height (*H*). Calls *TControl.Init* and adds to the list box object's *Attr.Style* field the Windows style constant *lbs_Standard*, which provides the scroll bar with:

- A border (*ws_Border*)
- A vertical scroll bar (*ws_VScroll*)
- Automatic alphabetic sorting of list items (*lbs_Sort*)
- Parent window notification upon selection (*lbs_Notify*)

These styles can be overridden in a descendant class, or in the *Init* constructor of the list box's parent window object.

AddString

```
function AddString(AString: PChar): Integer; virtual;
```

Override: Sometimes

Adds *AString* as a list item in the list box object and returns the item's position index (starting at zero) or a negative value in the case of an error. The list items are automatically sorted unless the style *lbs_Sort* is removed from the list box object's *Attr.Style* field before creation.

ClearList

```
procedure ClearList; virtual;
```

Override: Sometimes

Removes all of the list items from the list box.

DeleteString

```
function DeleteString(Index: Integer): Integer; virtual;
```

Override: Sometimes

Removes the list item at the position index (starting at zero) passed in *Index*. *DeleteString* returns the number of remaining list items or a negative value in the case of an error.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Never

Returns the name of *TListBox*'s window class, 'ListBox'.

GetCount

```
function GetCount: Integer; virtual;
```

Override: Seldom

Returns the number of list items in the list box or a negative value in the case of an error.

GetMsgID

```
function GetMsgID(AMsg: TMsgName): Word; virtual;
```

Translates list box messages for use by *TComboBox* objects.

GetSelIndex

```
function GetSelIndex: Integer; virtual;
```

Override: Seldom

Returns the position index (starting at zero) of the currently selected list item or a negative value if no item is selected.

GetSelString

```
function GetSelString(AString: PChar; MaxChars: Integer): Integer;
  virtual;
```

Override: Seldom

Retrieves in *AString* the currently selected list item, as long as it is no longer than *MaxChars*. *GetSelString* returns the string length or a negative value in the case of an error.

GetString

```
function GetString(AString: PChar; Index: Integer): Integer; virtual;
```

Override: Seldom

Retrieves in *AString* the list item at the position index (starting at zero) passed in *Index* and returns the string length or a negative value in the case of an error.

GetStringLen

```
function GetStringLen(Index: Integer): Integer; virtual;
```

Override: Seldom

Returns the string length of the list item at the position index passed in *Index* or a negative value in the case of an error.

InsertString

```
function InsertString(AString: PChar; Index: Integer): Integer; virtual;
```

Override: Sometimes

Inserts *AString* as a list item in the list box at the position index passed in *Index* and returns the item's actual position (starting at zero) or a negative value in the case of an error. The list box items are not resorted. If *Index* is –1, the string is appended to the end of the list.

SetSelIndex

```
function SetSelIndex(Index: Integer): Integer; virtual;
```

Override: Seldom

Forces the selection of the list item at the position index (starting at zero) passed in *Index*. If *Index* is –1, the list box is cleared of any selection. *SetSelIndex* returns a negative number in the case of an error.

SetSelString

```
function SetSelString(AString: PChar; AIndex: Integer): Integer;
  virtual;
```

Override: Seldom

Forces the selection of the first list item matching the text passed in *AString* that appears beyond the position index (starting at zero) passed in *AIndex*. If *AIndex* is –1, the entire list is searched from the beginning. *SetSelString* returns the position index of the newly selected item or a negative value in the case of an error.

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word;
  virtual;
```

Override: Sometimes

Transfers the list of entries and selected item(s) to or from the transfer record pointed to by *DataPtr*. If *TransferFlag* is *tf_GetData*, the list box's data is transferred to the memory location. If *TransferFlag* is *tf_SetData*, the list box is loaded with the data from the memory location. *Transfer* returns the number of bytes transferred. If *TransferFlag* is *tf_SizeData*, *Transfer* returns the size of the transfer data.

The nature of the record varies somewhat, depending on whether the box allows multiple items to be selected. The first item transferred is always a pointer to a collection of strings which are the list box's entries. For single-selection list boxes, that pointer is followed by an integer index to the selected item. For multiple-selection list boxes, the collection pointer is followed by a pointer to a *TMultiSelRec* record, which contains an array of integers, each indicating a selected item.

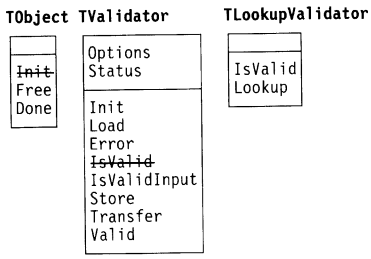
Typical transfer records for list boxes look like this:

```
type
  TListBoxXferRec = record                          { single selection }
    Strings: PStrCollection;
    Selection: Integer;
  end;

  TMultiListXferRec = record                        { multiple selection }
    Strings: PStrCollection;
    Selections: PMultiSelRec;
  end;
```

PMultiSelRec is a record defined in the *ODialogs* unit. The record must be allocated using the *AllocMultiSel* function and freed by *FreeMultiSel* after the transfer. A **nil** pointer indicates that no entries are selected.

TLookupValidator — Validate



A lookup validator compares the string typed by a user with a list of acceptable values. *TLookupValidator* is an abstract validator type from which you can derive useful lookup validators. You will never create an instance of *TLookupValidator*. When you create a lookup validator type, you need to specify a list of valid items and override the *Lookup* method to return *True* only if the user input matches an item in that list. One example of a working descendant of *TLookupValidator* is *TStringLookupValidator*.

Methods

IsValid

```
function IsValid(const S: string): Boolean; virtual;
```

Override: Seldom

Calls *Lookup* to find the string *S* in the list of valid input items. Returns *True* if *Lookup* returns *True*, meaning *Lookup* found *S* in its list; otherwise, returns *False*.

See also: *TLookupValidator.Lookup*

Lookup

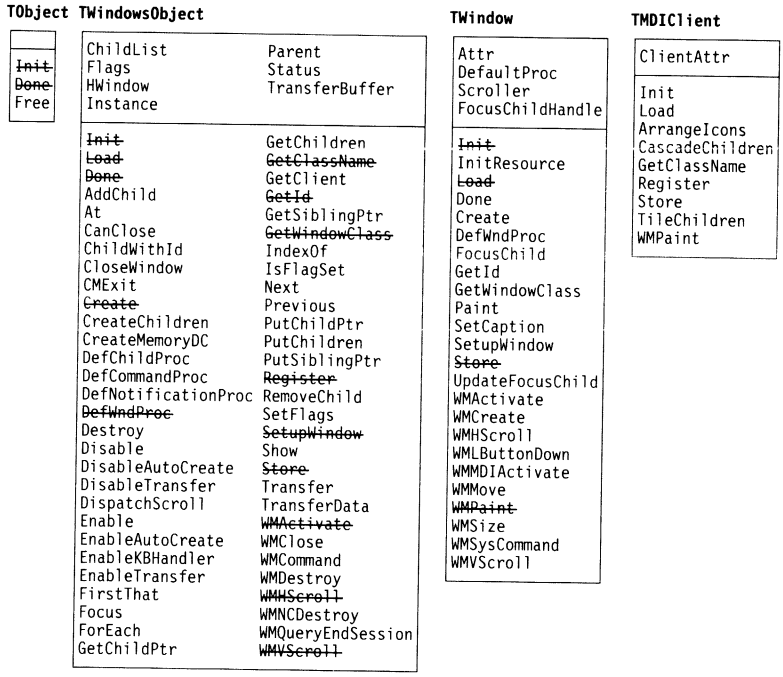
```
function Lookup(const S: string): Boolean; virtual;
```

Override: Often

Searches for the string *S* in the list of valid entries and returns *True* if it finds *S*; otherwise, returns *False*. *TLookupValidator*'s *Lookup* is an abstract method that always returns *False*. Descendant lookup validator types must override *Lookup* to perform a search based on the actual list of acceptable items.

TMDIClient

OWindows



Multiple Document Interface (MDI) client windows, represented by TMDIClient, are windows that manage the MDI child windows of an

MDI-compliant application. TMDIClient's methods are concerned with managing MDI child windows.

In earlier versions of ObjectWindows, *TMDIClient* descended from *TControl*. In the current version, *TMDIClient* descends directly from *TWindow*.

Field

ClientAttr

```
ClientAttr: TClientCreateStruct;
```

ClientAttr holds a record of the MDI client window's attributes. *TClientCreateStruct* is defined as:

```
type
  PClientCreateStruct = ^TClientCreateStruct;
  TClientCreateStruct = record
    hWindowMenu: THandle;
    idFirstChild: Word;
  end;
```

Methods

Init

```
constructor Init(AParent: PMDIWindow);
```

Override: Seldom

Constructs the MDI client window object with *AParent* as the parent window. *Init* sets the fields of *ClientAttr*. Calls the *Init* constructor inherited from *TWindow*, then adds the style *ws_ClipChildren* to *Attr.Style*. In addition, *Init* removes the client window from its parent's child-window list, so it is not treated as other child windows, such as list boxes and buttons.

See also: *TWindow.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads an MDI client window from the stream *S* by first calling *TControl.Load* and then reading the additional field (*ClientAttr*) introduced by *TMDIClient*.

See also: *TControl.Load*

ArrangeIcons

```
procedure ArrangeIcons; virtual;
```

Override: Seldom

Arranges the minimized MDI child windows into a neat row at the bottom of the MDI client window.

CascadeChildren

```
procedure CascadeChildren; virtual;
```

Override: Seldom

Sizes and arranges all of the non-minimized MDI child windows within the MDI client window so as to overlap and display the title bar of each one.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Never

Returns *TMDIClient*'s window class name, 'MDIClient'.

Register

```
function Register: Boolean; virtual;
```

Returns *True* because the MDI client window class is preregistered by Windows.

See also: *TWindowsObject.Register*

Store

```
procedure Store(var S: TStream);
```

Stores the MDI client window on the stream *S* by first calling *TControl.Store* and then writing the additional field (*ClientAttr*) introduced by *TMDIClient*.

See also: *TControl.Store*

TileChildren

```
procedure TileChildren; virtual;
```

Override: Seldom

Sizes and arranges all of the nonminimized MDI child windows within the MDI client window so as to take up all the available space without overlapping.

WMPaint

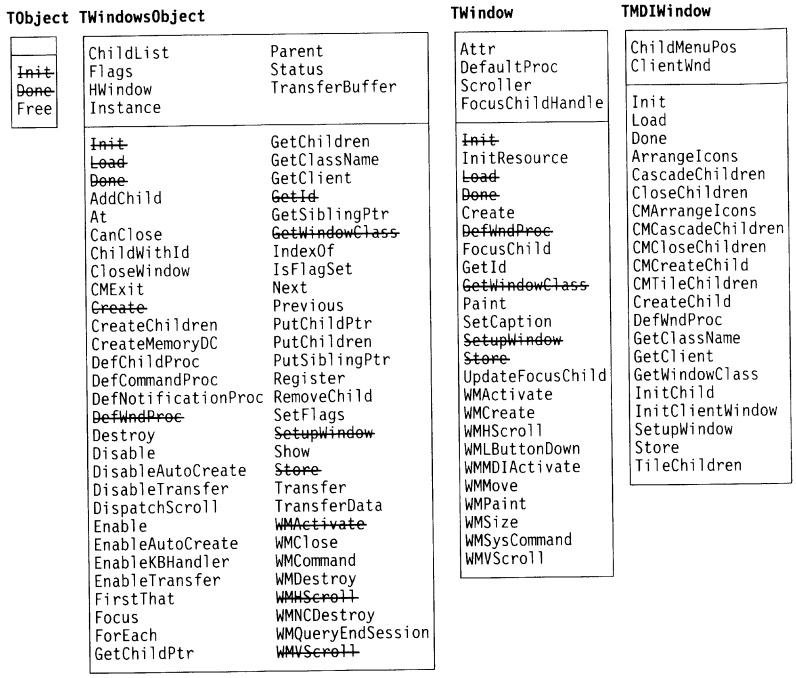
```
procedure WMPaint(var Msg: TMessage); virtual wm_First + wm_Paint;
```

Override: Never

Calls *DefWndProc* to paint the window, since it's a standard Windows class.

TMDIWindow

OWindows



Multiple Document Interface (MDI) frame windows, represented by *TMDIWindow*, are overlapped windows that serve as the main window of MDI-compliant applications. One major feature of *TMDIWindow* objects is that they own a *TMDIClient* object and store it in the *ClientWnd* field. Another feature is the child window menu that offers options for manipulating the application's MDI child windows. This window is automatically modified to reflect all displayed MDI child windows.

Fields

ChildMenuPos

```
ChildMenuPos: Integer;
```

Read/write

ChildMenuPos specifies an index identifying the position of the child window management menu. The index counts only top-level menu items and the top-left item is at position zero.

ClientWnd

```
ClientWnd: PMDIClient;
```

Read only

ClientWnd points to the MDI frame window's MDI client window, an object instance of *TMDIClient*.

Methods

Init

```
constructor Init(ATitle: PChar; AMenu: HMenu);
```

Override: Often

Constructs an MDI frame window object using the caption passed in *ATitle* and the menu passed in *AMenu*. MDI frame windows are required to have menus. An MDI frame window has no parent window and must be the main window of the application. As a default, *Init* sets *ChildMenuPos* to zero, indicating that the child window menu is the top-left menu. To modify *ChildMenuPos*, override *TMDIWindow.Init* in your descendant types. For example:

```
constructor MyMDIWindow.Init(ATitle: PChar; AMenu: HMenu);
begin
  TMDIWindow.Init(ATitle, AMenu);
  ChildMenuPos := 3;
end;
```

See also: *TMDIWindow.InitClientWindow*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads an MDI frame window from the stream *S* by first calling *TWindow.Load* and then getting and reading the additional fields (*ClientWnd* and *ChildMenuPos*) introduced by *TMDIWindow*.

See also: *TWindow.Load*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Disposes of the MDI client window object stored in *ClientWnd* before calling the *Done* destructor inherited from *TWindow* to dispose of the MDI frame window object.

See also: *TWindow.Done*

ArrangeIcons

```
procedure ArrangeIcons;
```

Override: Seldom

Arranges the minimized MDI child windows into a neat row at the bottom of the MDI client window. Calls *ClientWnd^.ArrangeIcons*.

See also: *TMDIClient.ArrangeIcons*

CascadeChildren

```
procedure CascadeChildren;
```

Override: Seldom

Sizes and arranges all of the nonminimized MDI child windows within the MDI client window so as to overlap and display the title bar of each one. Calls *ClientWnd^.CascadeChildren*.

See also: *TMDIClient.CascadeChildren*

CloseChildren

```
procedure CloseChildren;
```

Override: Seldom

Destroys all of the created MDI child windows for which *CanClose* returns *True*.

See also: *TMDIClient.CloseChildren*

CMArrangeIcons

```
procedure CMArrangeIcons(var Msg: TMessage); virtual cm_First + cm_ArrangeIcons;
```

Override: Seldom

Responds to a menu selection with an ID of *cm_ArrangeIcons* by calling *ArrangeIcons*.

See also: *TMDIWindow.ArrangeIcons*

CMCascadeChildren

```
procedure CMCascadeChildren(var Msg: TMessage); virtual cm_First +
  cm_CascadeChildren;
```

Override: Seldom

Responds to a menu selection with an ID of *cm_CascadeChildren* by calling *CascadeChildren*.

See also: *TMDIWindow.CascadeChildren*

CMCloseChildren

```
procedure CMCloseChildren(var Msg: TMessage); virtual cm_First +
  cm_CloseChildren;
```

Override: Seldom

Responds to a menu selection with an ID of *cm_CloseChildren* by calling *CloseChildren*.

See also: *TMDIWindow.CloseChildren*

CMCreateChild

```
procedure CMCreateChild(var Msg: TMessage); virtual cm_First + cm_CreateChild;
```

Override: Never

Responds to a menu selection with a menu ID of *cm_CreateChild* by calling *CreateChild* to produce a new child window.

See also: *TMDIWindow.CreateChild*

CMTileChildren

```
procedure CMTileChildren(var Msg: TMessage); virtual cm_First + cm_TileChildren;
```

Override: Seldom

Responds to a menu selection with an ID of *cm_TileChildren* by calling *TileChildren*.

See also: *TMDIWindow.TileChildren*

CreateChild

```
function CreateChild: PWindowsObject; virtual;
```

Constructs and creates a new MDI child window by calling *InitChild* and *MakeWindow*. You need not override *CreateChild* to accommodate descendant MDI child window types, as is true with *TMDIWindow.InitChild*. *CreateChild* returns a pointer to the new MDI child window.

See also: *TMDIWindow.InitChild, TApplication.MakeWindow*

DefWndProc

```
procedure DefWndProc(var Msg: TMessage); virtual;
```

Overrides *TWindow*'s default Windows message processing by calling the Windows function *DefFrameProc* rather than *DefWindowProc*.

See also: *TWindow.DefWndProc*

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Sometimes

Returns the name of *TMDIWindow*'s window class name, 'TurboMDIWindow'.

GetClient

```
function GetClient: PMDIClient; virtual;
```

Override: Never

Returns a pointer to the MDI client window stored in *ClientWnd*.

GetWindowClass

```
procedure GetWindowClass(var AWndClass: TWndClass); virtual;
```

Override: Sometimes

Modifies the default window class record and passes it back in *AWndClass*. *GetWindowClass* sets the style field to zero to remove the styles set by *TWindow.GetWindowClass*.

See also: *TWindow.GetWindowClass*

InitChild

```
function InitChild: PWindowsObject; virtual;
```

Override: Often

Constructs an MDI child window object (*TWindow*) with a caption of 'MDI Child' and returns a pointer to it. If you define an MDI child window type descending from *TWindow*, override *TMDIWindow.InitChild* to construct a window of your new MDI child window type. For example:

```
function MyMDIWindow.InitChild: PWindowsObject;
begin
  InitChild := New(PMyMDIChild, Init(@Self, 'Untitled Window'));
end;
```

See also: *TMDIWindow.CreateChild*

InitClientWindow

```
procedure InitClientWindow; virtual;
```

Override: Sometimes

Constructs the MDI client window as a *TMDIClient* object and stores it in *ClientWnd*.

SetupWindow

```
procedure SetupWindow; virtual;
```

Override: Often Constructs the MDI client window (*ClientWnd*) object's corresponding window element by calling *InitClientWindow* and creates it by calling *MakeWindow*. If you override *SetupWindow* in a descendant type, be sure to call *TMDIWindow.SetupWindow* explicitly.

See also: *TMDIWindow.InitClientWindow, TApplication.MakeWindow*

Store

```
procedure Store(var S: TStream);
```

Stores the MDI frame window on the stream *S* by first calling *TWindow.Store* and then putting and writing the additional fields (*ClientWnd* and *ChildMenuPos*) introduced by *TMDIWindow*.

See also: *TWindow.Store*

TileChildren

```
procedure TileChildren;
```

Override: Seldom Sizes and arranges all of the nonminimized MDI child windows within the MDI client window so as to take up all the available space without overlapping. Calls *ClientWnd^.TileChildren*.

See also: *TMDIClient.TileChildren*

TMessage type — OWindows

Declaration

```
TMessage = record
  Receiver: HWnd;
  Message: Word;
  case Integer of
    0: (WParam: Word;
        LParam: Longint;
        Result: Longint);
    1: (WParamLo: Byte;
        WParamHi: Byte;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
end;
```

Function The message processing loop in *TApplication* packages Windows message information into *TMessage* records before passing the information along to the appropriate message-response method.

See also *TApplication.MessageLoop*

TMultiSelRec type — ODialogs

Declaration

```
TMultiSelRec = record
  Count: Integer;
  Selections: array[0..0] of Integer;
end;
```

TMultiSelRec holds a list of selected items for transfer to or from a multiple-selection list box. *Count* indicates the number of selected items, and *Selections* is an open-ended array of integers. Using *AllocMultiSel*, you can allocate a record with enough selection items to accommodate as many selected items as the list box has.

See also *AllocMultiSel*, *FreeMultiSel*

TObject — Objects

TObject

TObject is the starting point of the ObjectWindows object hierarchy. As the base object, it has no parents but many descendants. All of the ObjectWindows standard objects are ultimately derived from *TObject*. Any object that uses the ObjectWindows streams facilities *must* trace its ancestry back to *TObject*.

Methods

Init

```
constructor Init;
```

Allocates space on the heap for the object. Called by all derived objects' constructors.

Free

```
procedure Free;
```

Disposes of the object and calls the *Done* destructor.

Done

```
destructor Done; virtual;
```

Performs the necessary cleanup and disposal for dynamic objects.

TPaintStruct type — WinTypes

Declaration

```
TPaintStruct = record
  hdc: HDC;
  fErase: Bool;
  rcPaint: TRect;
  fRestore: Bool;
  fIncUpdate: Bool;
  rgbReserved: array[0..15] of Byte;
end;
```

Function The *TPaintStruct* record holds information used by an application for painting the client areas of its windows. Most of the information is reserved for internal use by Windows, but several fields can be used by the user.

The *hdc* field is the handle of the display context on which the painting is to occur. *fErase* indicates whether the background needs to be redrawn; a nonzero value indicates that the application should redraw the background. *rcPaint* defines the rectangle in which painting is to take place.

All the other fields are reserved for internal Windows use.

TPicResult type — Validate

Declaration

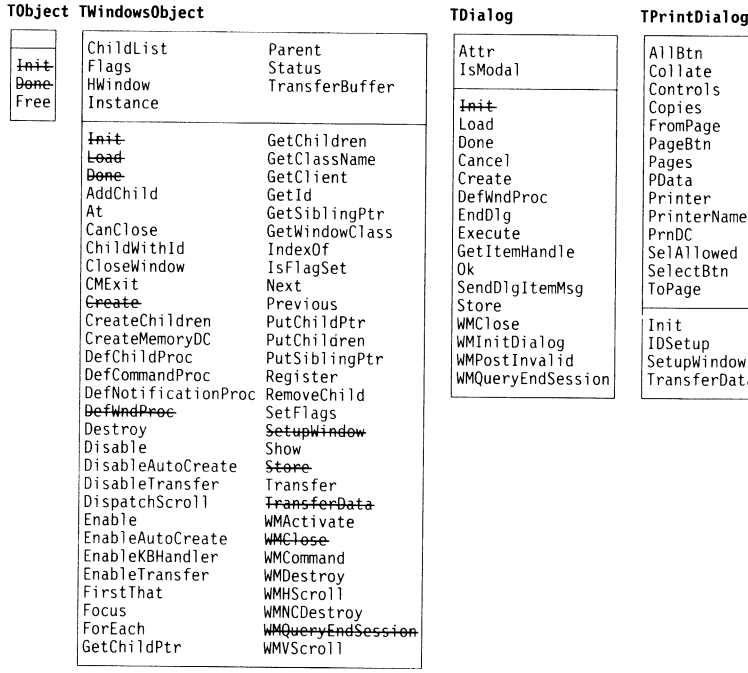
```
TPicResult = (prComplete, prIncomplete, prEmpty, prError, prSyntax, prAmbiguous,
  prIncompNoFill);
```

Function *TPicResult* is the result type returned by the *Picture* method of *TPXPictureValidator*.

See also *TPXPictureValidator.Picture*

TPrintDialog object

OPrinter



The print dialog box gives the user the ability to customize a particular print job by choosing such things as pages to print, number of copies, or what printer to use. *TPrintDialog* is the default print dialog box object, but you can insert your own custom dialog box by overriding the printer object's *InitPrintDialog* method.

Fields

AllBtn

```
AllBtn: PRadioButton;
```

AllBtn is one of a set of three radio buttons. If checked, *AllBtn* indicates that the printer should print all pages in the document.

See also: *TPrintDialog.SelectBtn*, *TPrintDialog.PageBtn*

Collate

```
Collate: PCheckBox;
```

Points to a check box in the dialog box that indicates whether the user wants the output collated.

Controls `Controls: PCollection;`

Used internally by the print dialog box.

Copies `Copies: PEdit;`

Copies enables the user to specify the number of copies of the document to print.

FromPage `FromPage: PEdit;`

If the user chooses to print a range of pages, *FromPage* holds the number of the first page to print.

See also: *TPrintDialog.PageBtn*, *TPrintDialog.ToPage*

PageBtn `PageBtn: PRadioButton;`

PageBtn is one of a set of three radio buttons. If checked, the user wants to print a selected range of page numbers.

See also: *TPrintDialog.AllBtn*, *TPrintDialog.FromPage*, *TPrintDialog.SelectBtn*, *TPrintDialog.ToPAge*

Pages `Pages: Integer;`

Pages is the total number of pages in the document.

PData `PData: PPrintDialogRec;`

PData points to a record of type *TPrintDialogRec*. The print dialog box uses the record as its transfer buffer.

See also: *TPrintDialogRec* type

Printer `Printer: PPrinter;`

Printer points to the printer object associated with the print dialog box.

PrinterName `PrinterName: PStatic;`

Points to a static text control holding the name of the currently selected printer.

PrnDC `PrnDC: HDC;`

PrnDC is the handle of the device context to use for printing.

SelAllowed `SelAllowed: Boolean;`

If *True*, indicates that the printer device supports printing only the selected text in the document. *SelectBtn* is only enabled if *SelAllowed* is *True*.

See also: *TPrintDialog.SelectBtn*

SelectBtn

```
SelectBtn: PRadioButton;
```

SelectBtn is one of a set of three radio buttons. If checked, the user wants to print the text currently selected in the document.

See also: *TPrintDialog.SelAllowed*

ToPage

```
ToPage: PEdit;
```

If the user chooses to print a range of pages, *ToPage* holds the number of the last page to print.

See also: *TPrintDialog.PageBtn*, *TPrintDialog.FromPage*

Methods

Init

```
constructor Init(AParent: PWindowsObject; Template: PChar; APrnDC: HDC; APages:
  Integer; APrinter: PPrinter; ASelAllowed: Boolean; var Data: TPrintDialogRec);
```

Constructs a print dialog box by first calling the *Init* constructor inherited from *TDialog*, then setting fields according to the values of the passed parameters. *Init* then constructs control objects for the dialog box by calling *InitResource* for each.

IDSetup

```
procedure IDSetup(var Msg: TMessage); virtual id_First + id_Setup;
```

Responds to the user pressing the Setup button by invoking the printer setup dialog box. By default, the printer setup dialog box is an instance of type *TPrinterSetupDialog*.

See also: *TPrinterSetupDialog* object

SetupWindow

```
procedure SetupWindow; virtual;
```

Initializes the dialog box by first calling the *SetupWindow* method inherited from *TDialog*, then retrieving the printer device name from the printer object and setting the *PrinterName* field to that value.

TransferData

```
procedure TransferData(Direction: Word); virtual;
```

Overrides the inherited *TransferData* method to set control values based on the values in *PData* if *Direction* is *tf_SetData*, or set *PData* fields based on control values if *Direction* is *tf_GetData*. *TransferData* does not just set or read controls, as there is not a one-to-one correspondence between *PData* fields and print dialog box controls.

TPrintDialogRec type — OPrinter

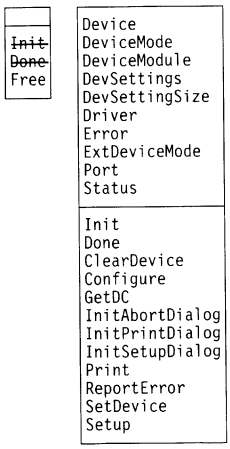
Declaration

```
TPrintDialogRec = record
  drStart: Integer;
  drStop: Integer;
  drCopies: Integer;
  drCollate: Boolean;
  drUseSelection: Boolean;
end;
```

Function Print dialog objects use records of type *TPrintDialogRec* as transfer buffers. The *drStart* and *drStop* fields represent the first and last pages to print, respectively. *drCopies* indicates the number of copies to print. *drCollate* tells the printer to collate the copies if *drCopies* called for more than one. *drUseSelection* tells the printer to print the selected text rather than the text indicated by *drStart* and *drStop*.

TPrinter — OPrinter

TObject TPrinter



The *TPrinter* object type represents an encapsulation of the printer driver system of Windows. To print on or configure a printer, initialize an instance of *TPrinter*.

Fields

Device

```
Device: PChar;
```

Pointer to the name of the current device. If **nil**, the object is not currently connected to a device.

DeviceMode

```
DeviceMode: TDeviceMode;
```

Function variable that contains the address of the *DeviceMode* function of the currently associated printer. This variable is used during a call to *Configure* if the device does not support *ExtDeviceMode*.

DeviceModule

```
DeviceModule: THandle;
```

The handle to the currently associated printer driver.

DeviceSettings

```
DeviceSettings: PDevMode;
```

Pointer to a local copy of the printer device settings (also called the environment). This is used only if the current printer device supports *ExtDeviceMode*.

DeviceSettingSize

```
DeviceSettingSize: Integer;
```

Amount of memory allocated to *DeviceSettings*.

Driver

```
Driver: PChar;
```

Pointer to the name of the current driver. If **nil**, the object is not currently connected to a driver.

Error

```
Error: Integer;
```

Error code returned by GDI during printing. This value is initialized during a call to *Print*.

ExtDeviceMode

```
ExtDeviceMode: TExtDeviceMode;
```

Function variable that contains the address of the *ExtDeviceMode* function of the currently associated printer device. The address is **nil** if the driver does not support this Windows 3.0-specific routine (that is, *@ExtDeviceMode* = **nil** is *True*). This variable is used during a call to *Configure*.

Port

```
Port: PChar;
```

Pointer to the name of the port to which the current printer is attached. If **nil**, the object is not connected to a printer.

Status

```
Status: Integer;
```

Current status of the printer driver.

Methods

Init

```
constructor Init;
```

Creates an instance of *TPrinter* associated with the default printer. To change the printer *TPrinter* call *SetDevice* after the object has been initialized or call *Setup* to allow the user to select the new device through a dialog.

Done

```
destructor Done; virtual;
```

Override: Seldom

Frees the resources allocated to *TPrinter*.

ClearDevice

```
procedure ClearDevice;
```

Disassociates the device with the current printer. Called by *SetDevice* and *Done*. Changes the current status of the printer to *pf_Unassociated*, which will cause the object to ignore all calls to *Print* until the object is reassociated with a printer.

Configure

```
procedure Configure(Window: PWindowsObject);
```

Calls the device driver to configure the current printer. If the printer driver supports *ExtDeviceMode*, a local copy of the printer settings are kept, and the printer defaults are not modified. If the printer only supports *DeviceMode*, the global defaults for the printer will be modified.

GetDC

```
function GetDC: HDC; virtual;
```

Override: Seldom

Returns a device context for the currently associated printer. Returns zero if the object is in an invalid state or the printer is associated to an inactive port (that is, associated with the port "None").

InitAbortDialog

```
function InitAbortDialog(Parent: PWindowsObject; Title: PChar): PDialog; virtual;
```

Override: Seldom

Called by *Print* at the beginning of a print job. The return result is created as a modeless dialog box. By default, returns an instance of *TPrinterAbortDlg*. Canceling this dialog box cancels the print job.

Can be overridden to return a customized dialog displayed during printing.

InitPrintDialog

```
function InitPrintDialog(Parent: PWindowsObject; PrnDC: HDC; Pages: Integer;
  SelAllowed: Boolean; var Data: TPrintDialogRec): PDialog; virtual;
```

Override: Seldom

Called by *Print* to return a print dialog box if the printout object indicates that it supports printing of selected pages. The print dialog box lets the user specify whether to print all pages, selected text, or a range of pages. By default, *InitPrintDialog* returns an instance of the type *TPrintDialog*.

InitSetupDialog

```
function InitSetupDialog(Parent: PWindowsObject): PDialog; virtual;
```

Override: Seldom

Called by *Setup* to return a printer-setup dialog box. The result is executed on the application as a modal dialog box. By default, returns an instance of *TPrinterSetupDlg*. Can be overridden to return a customized printer setup dialog.

Print

```
function Print(ParentWin: PWindowsObject; PrintOut: PPrintOut): Boolean;
```

Renders the given printout object on the associated printer device. Displays an abort dialog box while printing and displays any errors encountered during printing.

ReportError

```
procedure ReportError(PrintOut: PPrintOut); virtual;
```

Override: Sometimes

Print calls *ReportError* if it encounters an error. By default it will bring up the system message box with an error string created from the default string table values 32512 – 32519. This method can be overridden to show a custom error dialog box.

SetDevice

```
procedure SetDevice(ADevice, ADriver, APort: PChar);
```

Changes the printer device association. *Setup* calls *SetDevice* to change the association interactively. The valid parameters to this method can be found in the [devices] section of the WIN.INI file.

Entries in the [devices] section have the following format:

```
<device name>=<driver>, <port> {, <port>}
```

where the port can be repeated any number of times.

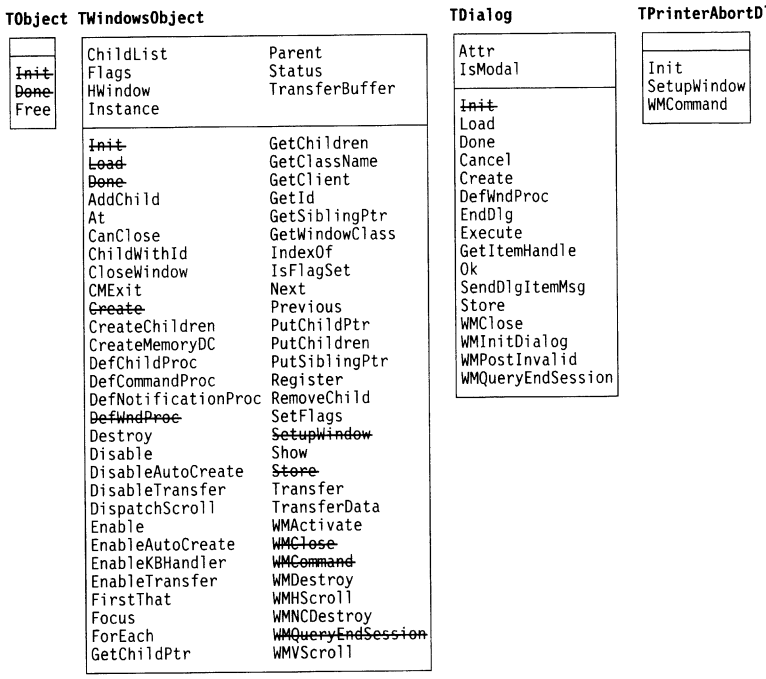
Setup

```
procedure Setup(Parent: PWindowsObject);
```

Call this method when you want the user to select and/or configure the currently associated printer. Calls *InitSetupDialog* to return the printer-setup dialog box shown to the user.

TPrinterAbortDlg

OPrinter



This is the object type of the default printer-abort dialog box. This dialog is initialized to display the title of the current printout as well as the device and port currently being used to print.

TPrinterAbortDlg expects to have three static text controls with control IDs of 101 for the title, 102 for the device, and 103 for the port, respectively. These controls must have '%s' somewhere in their text strings which get replaced by the title, device, and port, respectively. The position and tab order of the controls in the dialog box are not important.

Methods

Init

```
constructor Init(AParent: PWindowsObject; Template, Title, Device, Port: PChar);
```

Constructs an abort dialog box that displays the given title, device, and port in the dialog box along with a Cancel button.

SetupWindow

```
procedure SetupWindow; virtual;
```

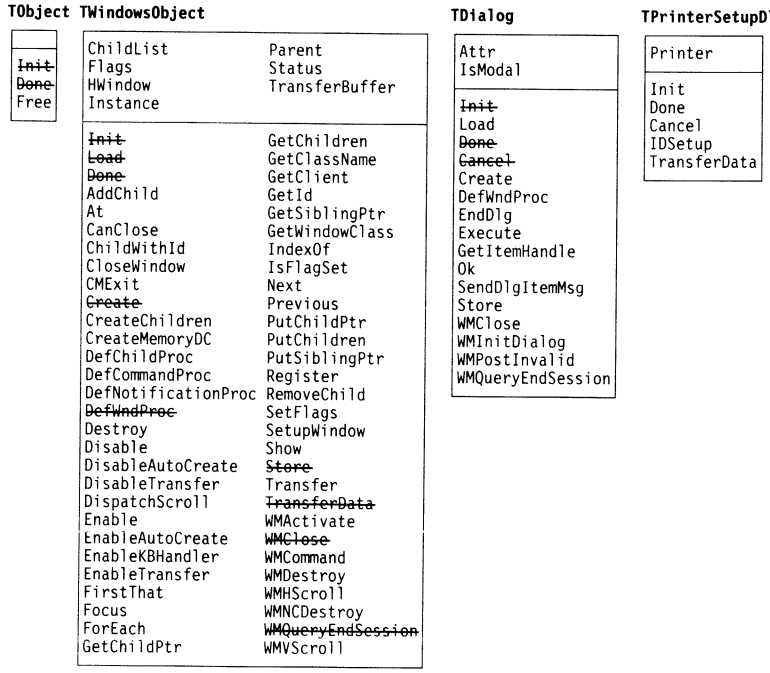
Override: Seldom

Used internally to associated objects with the dialog resource template so the title, device, and port can be filled in.

WMCommand `procedure WMCommand(var Msg: TMessage); virtual wm_First + wm_Command;`

Override: Seldom Used internally to handle the Cancel button.

TPrinterSetupDlg — OPrinter



This is the default printer-setup dialog box. It resembles the printer-setup dialog box in the Windows-hosted IDE. *TPrinterSetupDlg* expects to have a combo box for the list of valid devices and a Setup push button. These have control IDs of 101 and 102, respectively. It should also have OK and Cancel buttons.

Field

Printer `Printer: PPrinter;`

Pointer to the current printer being modified by the dialog.

Methods

Init

```
constructor Init(AParent: PWindowsObject; TemplateName: PChar; APrinter:
  PPrinter);
```

Constructs a printer-setup dialog that modifies the given printer.

Done

```
destructor Done; virtual;
```

Override: Seldom

Deallocates the resources allocated to the object.

Cancel

```
procedure Cancel(var Msg: TMessage); virtual id_First + id_Cancel;
```

Override: Never

Used internally to restore the previous state of the printer if the user presses Setup followed by Cancel.

IDSetup

```
procedure IDSetup(var Msg: TMessage); virtual id_First + id_Setup;
```

Override: Never

Handles the Setup button press. Modifies the given printer and calls its *Configure* method.

TransferData

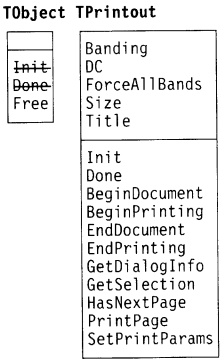
```
procedure TransferData(TransferFlag: Word); virtual;
```

Override: Never

Used internally to transfer the data out of the dialog box and modify the given printer directly.

TPrintout OPrinter

TObject TPrintout



This object is used in conjunction with a *TPrinter* object to print information on a printer. This object type is abstract, meaning it cannot be used to print anything by itself. A descendent of *TPrinter* must be created and its *PrintPage* method overridden to print what is desired.

Fields

Banding

```
Banding: Boolean;
```

If *True*, the printout is banded and the *PrintPage* method is called once for every band. Otherwise, the *PrintPage* method is only called once for every page. Banding a printout is more memory- and time-efficient than not banding. By default, this field is *False*.

DC

```
DC: HDC;
```

DC is the handle to the device context to use for printing.

ForceAllBands

```
ForceAllBands: Boolean;
```

Many device drivers do not provide all printer bands if both text and graphics are not performed on the first band (typically a text-only band). Leaving this *True* forces the printer driver to give all bands regardless of what calls are made in the *PrintPage* method. If *PrintPage* does nothing but display text, it is more efficient for this field to be *False*. By default, this field is *True*. This field only takes effect if *Banding* is *True*.

Size

```
Size: TPoint;
```

Size represents the size of the print area on the printout page.

Title

```
Title: PChar;
```

The current title used for the printout. By default, this title appears in the abort dialog box and as the name of the job in the Print Manager.

Methods

Init

```
constructor Init(ATitle: PChar);
```

Constructs an instance of *TPrintOut* with the given title.

Done

```
destructor Done; virtual;
```

Disposes of the resources allocated by the *Init* constructor.

BeginDocument

```
procedure BeginDocument(StartPage, EndPage: Integer; Flag: Word); virtual;
```

The printer object's *Print* method calls *BeginDocument* once before printing each copy of a document. The *Flags* field holds *pf_Banding* or *pf_Selection* to indicate that banding will be used or that selected text should be printed.

The default *BeginDocument* does nothing. Descendant objects can override *BeginDocument* to perform any initialization needed at the beginning of each copy of the document.

BeginPrinting

```
procedure BeginPrinting; virtual;
```

The printer object's *Print* method calls *BeginPrinting* once at the beginning of a print job, regardless of how many copies of the document will print.

The default *BeginPrinting* does nothing. Descendant objects can override *BeginPrinting* to perform any initialization needed before printing.

EndDocument

```
procedure EndDocument; virtual;
```

The printer object's *Print* method calls *EndDocument* after each copy of the document finishes printing.

The default *EndDocument* does nothing. Descendant objects can override *EndDocument* to perform any needed actions at the end of each document.

EndPrinting

```
procedure EndPrinting; virtual;
```

The printer's *Print* method calls *EndPrinting* after all copies of the document finish printing.

The default *EndPrinting* does nothing. Descendant objects can override *EndPrinting* to perform any needed actions at the end of printing.

GetDialogInfo

```
function GetDialogInfo(var Pages: Integer): Boolean; virtual;
```

Retrieves the information needed to allow printing of selected pages in the document and returns *True* if page selection is possible. Use of *Pages* is optional, but if the page count is easy to determine, *GetDialogInfo* should set *Pages* to the number of pages in the document. Otherwise, *Pages* should be set to zero, and printing continues until *HasNextPage* returns *False*.

GetSelection

```
function GetSelection(var Start, Stop: Integer): Boolean; virtual;
```

Determines whether the document has selected text. If it does, the function returns *True* and *Start* and *Stop* point to the beginning and ending of the selected text, respectively. If *GetSelection* returns *False*, the print selection button in the print dialog box is disabled.

The default *GetSelection* just returns *False*. Descendant objects can override *GetSelection* to actually determine whether a selection exists.

HasNextPage

```
function HasNextPage: Boolean; virtual;
```

This method is called after every page. By default, it always returns *False*, indicating that only one page is to be printed. If the document contains more than one page this method needs to be overridden to return *True* while there are more pages to print.

PrintPage

```
procedure PrintPage(Page: Word; var Rect: TRect; Flags: Word); virtual;
```

Called for every page (or band, if *Banding* is *True*). Must be overridden to print the contents of the given page. The *Rect* and *Flags* parameters are used during banding to indicate the extent and the type of band currently requested from the driver (should be ignored if *Banding* is *False*). The *Size* parameter is the size, in device pixels, of the page being printed. *Page* is the number of the current page, and *DC* is the device context of the printer the overridden method should use in all GDI calls.

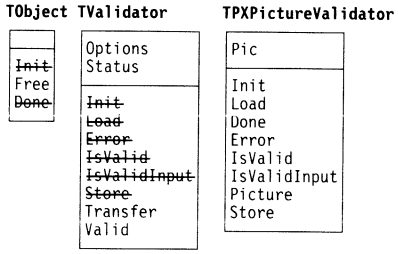
SetPrintParams

```
procedure SetPrintParams(ADC: HDC; ASize: TPoint): virtual;
```

Sets the *DC* and *Size* fields to *ADC* and *ASize*, respectively. This is the first printout method called by the printer object's *Print* method, providing the printout object with the information it will need to determine pagination and page count. If descendant objects override *SetPrintParams*, they must call the inherited method.

TPXPictureValidator

Validate



Picture validator objects compare user input with a picture of a data format to determine the validity of entered data. The pictures are compatible with the pictures Borland's Paradox relational database uses to control data entry. For a complete description of picture specifiers, see *TPXPictureValidator*'s *Picture* method.

T

Field

Pic

```
Pic: PString;
```

Points to a string containing the picture that specifies the format for data in the associated input line. The *Init* constructor sets *Pic* to a string passed as one of its parameters.

Methods

Init

```
constructor Init(const APic: string; AutoFill: Boolean);
```

Constructs a picture validator object by first calling the *Init* constructor inherited from *TValidator*, then allocating a copy of *APic* on the heap and setting *Pic* to point to it, then setting the *voFill* bit in *Options* if *AutoFill* is *True*.

See also: *TValidator.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a picture validator object from the stream *S* by first calling the *Load* constructor inherited from *TValidator*, then reading the value for the *Pic* field introduced by *TPXPictureValidator*.

See also: *TValidator.Load*

Done

```
destructor Done; virtual;
```

Disposes of the string pointed to by *Pic*, then disposes of the picture validator object by calling the *Done* destructor inherited from *TValidator*.

Error

```
procedure Error; virtual;
```

Executes a message box indicating an error in the picture format, and displaying the string pointed to by *Pic*.

IsValidInput

```
function IsValidInput(var S: string; SuppressFill: Boolean): Boolean;
  virtual;
```

Checks the string passed in *S* against the format picture specified in *Pic* and returns *True* if *Pic* is **nil** or *Picture* does not return *prError* for *S*; otherwise, returns *False*. The *SuppressFill* parameter overrides the value in *voFill* for the duration of the call to *IsValidInput*.

Because *S* is a **var** parameter, *IsValidInput* can modify its value. For example, if *SuppressFill* is *False* and *voFill* is set, the call to *Picture* returns a filled string based on *S*, so the image in the input line automatically reflects the format specified in *Pic*.

See also: *TPXPictureValidator.Picture*

IsValid

```
function IsValid(const S: string): Boolean; virtual;
```

Compares the string passed in *S* with the format picture specified in *Pic* and returns *True* if *Pic* is **nil** or if *Picture* returns *prComplete* for *S*, indicating that *S* needs no further input to meet the specified format.

See also: *TPXPictureValidator.Picture*

Picture

```
function Picture(var Input: string): TPicResult; virtual;
```

Formats the string passed in *Input* according to the format specified by the picture string pointed to by *Pic*. Returns *prError* if there is an error in the picture string or if *Input* contains data that cannot fit the specified picture. Returns *prComplete* if *Input* can fully satisfy the specified picture. Returns *prIncomplete* if *Input* contains data that fits the specified picture but not completely.

Table 21.26 shows the characters used in creating format pictures.

Table 21.26 Picture format characters

Type of character	Character	Description
Special	#	Accept only a digit
	?	Accept only a letter (case-insensitive)
	&	Accept only a letter, force to uppercase
	@	Accept any character
	!	Accept any character, force to uppercase
Match	;	Take next character literally
	*	Repetition count
	[]	Option
	{}	Grouping operators
	,	Set of alternatives
All others		Taken literally

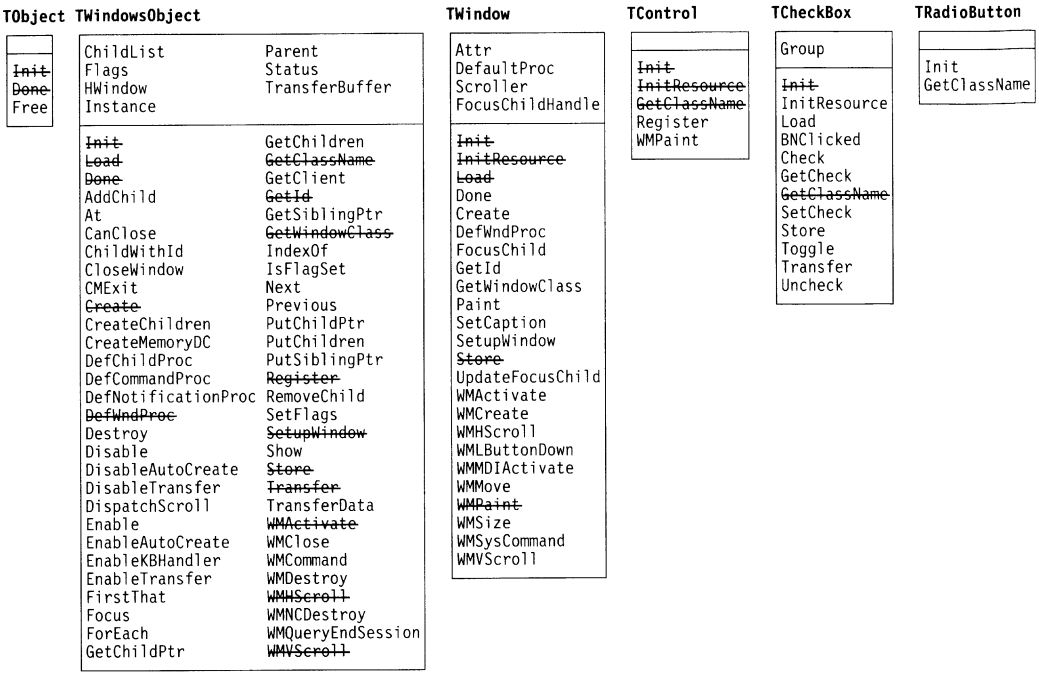
See also: *TPicResult* type

Store

```
procedure Store(var S: TStream);
```

Stores the picture validator object on the stream *S* by first calling the *Store* method inherited from *TValidator*, then writing the string pointed to by *Pic*.

TRadioButton



TRadioButton is an interface object that represents a corresponding radio button element in Windows. Radio buttons have two states: checked and unchecked. *TRadioButton* inherits its state management methods from its ancestor, *TCheckBox*. Optionally, a check box can be part of a group (*TGroupBox*) which visually and conceptually groups its controls.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; ATitle: PChar; X,Y,W,H:
  Integer; AGroup: PGroupBox);
```

Override: Sometimes

Constructs a radio button object with the passed parent window (*AParent*), control ID (*AnId*), associated text (*ATitle*), position relative to the origin of the parent window's client area (*X*, *Y*), width (*W*), height (*H*), and associated group box (*AGroup*). Sets the check box's *Attr.Style* field to *ws_Child* **or** *ws_Visible* **or** *bs_RadioButton*.

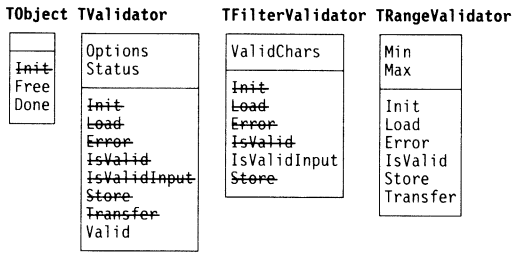
GetClassName

```
function GetClassName: PChar; virtual;
```

Calls the *GetClassName* method inherited from *TCheckBoxes* unless you're using BWCC controls, in which case it returns 'BorRadio'.

See also: *TCheckBoxes.GetClassName*

TRangeValidator — Validate



A range validator object determines whether the data typed by a user falls within a designated range of integers.

Fields

Max

```
Max: Longint;
```

Max is the highest valid long integer value for the input line.

Min

```
Min: Longint;
```

Min is the lowest valid long integer value for the input line.

Methods

Init

```
constructor Init(AMin, AMax: Longint);
```

Constructs a range validator object by first calling the *Init* constructor inherited from *TFilterValidator*, passing a set of characters containing the digits '0'..'9' and the characters '+' and '-'. Sets *Min* to *AMin* and *Max* to *AMax*, establishing the range of acceptable long integer values.

See also: *TFilterValidator.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a range validator object from the stream *S* by first calling the *Load* constructor inherited from *TFilterValidator*, then reading the *Min* and *Max* fields introduced by *TRangeValidator*.

See also: *TFilterValidator.Load*

Error `procedure Error; virtual;`

Displays a message box indicating that the entered value did not fall in the specified range.

IsValid `function IsValid(const S: string): Boolean; virtual;`

Converts the string *S* into an integer number and returns *True* if the result meets all three of these conditions:

- it is a valid integer number
- its value is greater than or equal to *Min*
- its value is less than or equal to *Max*

If any of those tests fails, *IsValid* returns *False*.

Store `procedure Store(var S: TStream);`

Stores the range validator object on the stream *S* by first calling the *Store* method inherited from *TFilterValidator*, then writing the *Min* and *Max* fields introduced by *TRangeValidator*.

See also: *TFilterValidator.Store*

Transfer
```
function Transfer(var S: String; Buffer: Pointer; Flag: TVTransfer): Word;
  virtual;
```

Incorporates the three functions, *DataSize*, *GetData*, and *SetData*, that a range validator can handle for its associated input line. Instead of setting and reading the value of the numeric input line by passing a string representation of the number, *Transfer* can use a *Longint* as its data record, which keeps your application from having to handle the conversion.

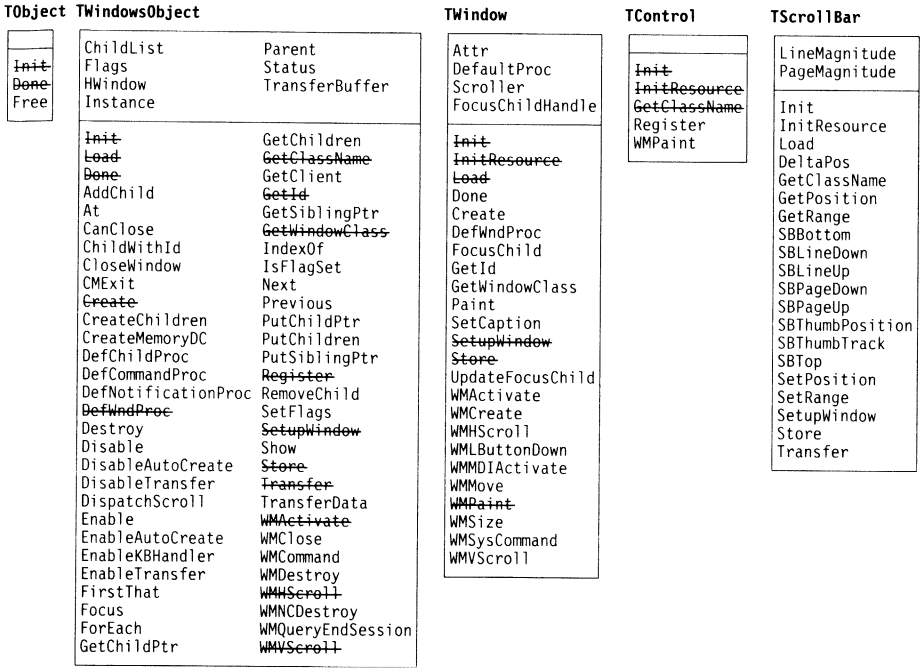
S is the input line's string value, and *Buffer* is the data record passed to the input line. Depending on the value of *Flag*, *Transfer* either sets *S* from the number in *Buffer*^ or sets the number at *Buffer* to the value of the string *S*. If *Flag* is *vtSetData*, *Transfer* sets *S* from *Buffer*. If *Flag* is *vtGetData*, *Transfer* sets *Buffer* from *S*. If *Flag* is *vtDataSize*, *Transfer* neither sets nor reads data.

Transfer always returns the size of the data transferred, in this case the size of a *Longint*.

See also: *TVTransfer* type

TScrollBar

ODialogs



TScrollBar objects represent standalone scroll bar controls, but not the scroll bars that are attached to windows. Scroll bars can be vertical (scrolls up and down) or horizontal (scrolls right and left). Most of *TScrollBar*'s methods are concerned with managing the scroll bar's thumb position and range.

One special feature of the type *TScrollBar* is the set of methods that automatically respond to the Windows scroll bar messages, *wm_HScroll* and *wm_VScroll*. The methods, such as *SBLineUp* and *SBPageDown*, are defined as notify-based methods. These methods automatically adjust the scroll bar's thumb position.

TScrollBar objects should never be placed in windows that have either the *ws_HScroll* or *ws_VScroll* styles in their attributes.

Fields

LineMagnitude

```
LineMagnitude: Integer;
```

Read/write

LineMagnitude is the number of range units to scroll the scroll bar when the user requests a small movement by clicking on the scroll bar's arrows.

Init sets *LineMagnitude* to 1 by default. *TScrollBar.InitWindow* sets the scroll range from zero to 100 by default.

See also: *TScrollBar.InitWindow*

PageMagnitude

```
PageMagnitude: Integer;
```

Read/write

PageMagnitude is the number of range units to scroll the scroll bar when the user requests a large movement by clicking in the scroll bar's scrolling area. *TScrollBar.Init* sets *PageMagnitude* to 10 by default. (The scroll range is set to from zero to 100 by default.)

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; X, Y, W, H:
  Integer; IsHScrollBar: Boolean);
```

Constructs and initializes a *TScrollBar* object with the given parent window (*AParent*), *AnID* as a control ID, a position of (*X*, *Y*), a width of *W* and a height of *H*. The scroll bar is horizontal (style *sbs_Horz*) if *IsHScrollBar* is *True* and vertical (style *sbs_Vert*) if it is *False*. If the supplied height for a horizontal scroll bar or the supplied width for a vertical scroll bar is zero, a standard value is used. *LineMagnitude* is initialized to 1 and *PageMagnitude* to 10.

InitResource

```
constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
```

Associates a scroll bar object with the control element in the resource specified by *ResourceID* by calling the *InitResource* constructor inherited from *TControl*, then setting *LineMagnitude* to 1 and *PageMagnitude* to 10.

See also: *TControl.InitResource*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a scroll bar control from the stream *S* by first calling *TControl.Load* and then reading the additional field (*IsHorizontal*, *LineMagnitude* and *PageMagnitude*) introduced by *TScrollBar*.

See also: *TControl.Load*

DeltaPos

```
function DeltaPos(Delta: Integer): Integer; virtual;
```

Override: Seldom

Changes the scroll bar's thumb position by the value passed in *Delta* by calling *SetPosition*. A negative value moves the thumb up or left. Returns the new thumb position.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Never

Returns the name of *TScrollBar*'s window class, 'Scrollbar'.

GetPosition

```
function GetPosition: Integer; virtual;
```

Override: Seldom

Returns the scroll bar's current thumb position.

See also: *TScrollBar.SetPosition*

GetRange

```
procedure GetRange(var LoVal, HiVal: Integer); virtual;
```

Override: Seldom

Retrieves the allowed range of scroll bar thumb positions in *LoVal* and *HiVal*.

See also: *TScrollBar.SetRange*

SBBottom

```
procedure SBBottom(var Msg: TMessage); virtual nf_First + sb_Bottom;
```

Override: Seldom

Sets the thumb position to the highest allowable value by calling *GetRange* and *SetPosition*. Called in response to a scroll-based message carrying the code *sb_Bottom*.

SBLineDown

```
procedure SBLineDown(var Msg: TMessage); virtual nf_First + sb_LineDown;
```

Override: Seldom

Moves the thumb position down or right by *LineMagnitude* by calling *DeltaPos*. Called in response to a scroll-based message carrying the code *sb_LineDown*.

SBLineUp

```
procedure SBLineUp(var Msg: TMessage); virtual nf_First + sb_LineUp;
```

Override: Seldom

Moves the thumb position up or left by *LineMagnitude* by calling *DeltaPos*. Called in response to a scroll-based message carrying the code *sb_LineUp*.

SBPageDown

```
procedure SBPageDown(var Msg: TMessage); virtual nf_First + sb_PageDown;
```

Override: Seldom

Moves the thumb position down or right by *PageMagnitude* by calling *DeltaPos*. Called in response to a scroll-based message carrying the code *sb_PageDown*.

SBPageUp

```
procedure SBPageUp(var Msg: TMessage); virtual nf_First + sb_PageUp;
```

Override: Seldom

Moves the thumb position up or left by *PageMagnitude* by calling *DeltaPos*. Called in response to a scroll-based message carrying the code *sb_PageUp*.

SBThumbPosition

```
procedure SBThumbPosition(var Msg: TMessage); virtual nf_First +
  sb_ThumbPosition;
```

Override: Seldom

Changes the thumb position to the position chosen by the user and passed in a scroll-based message carrying the code *sb_ThumbPosition* by calling *SetPosition*.

SBThumbTrack

```
procedure SBThumbTrack(var Msg: TMessage); virtual nf_First + sb_ThumbTrack;
```

Override: Sometimes

Changes the thumb position as the user drags the thumb by calling *SetPosition*. Called in response to a scroll-based message carrying the code *sb_ThumbTrack*.

SBTop

```
procedure SBTop(var Msg: TMessage); virtual nf_First + sb_Top;
```

Override: Seldom

Sets the thumb position to its minimum value by calling *GetRange* and *SetPosition*. Called in response to a scroll-based message carrying the code *sb_Top*.

SetPosition

```
procedure SetPosition(ThumbPos: Integer);
```

Override: Sometimes

Sets the scroll bar's thumb position to *ThumbPos*. If *ThumbPos* is higher than the maximum value, the thumb position is set to the maximum value. If *ThumbPos* is lower than the minimum value, the thumb is set to the minimum value.

See also: *TScrollBar.GetPosition*

SetRange

```
procedure SetRange(LoVal, HiVal: Integer); virtual;
```

Override: Seldom

Sets the scroll bar's allowable range of values from *LoVal* to *HiVal*.

See also: *TScrollBar.GetRange*

SetupWindow

```
procedure SetupWindow; virtual;
```

Override: Sometimes

Initializes the scroll bar by calling the *SetupWindow* method inherited from *TControl*, then calling *SetRange* to initialize the scroll bar's range to zero to 100.

See also: *TScrollBar.SetRange*

Store

```
procedure Store(var S: TStream);
```

Stores the scroll bar control on the stream *S* by first calling the *Store* method inherited from *TControl*, then writing the additional fields (*LineMagnitude* and *PageMagnitude*) introduced by *TScrollBar*.

See also: *TControl.Store*

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
```

Override: Sometimes

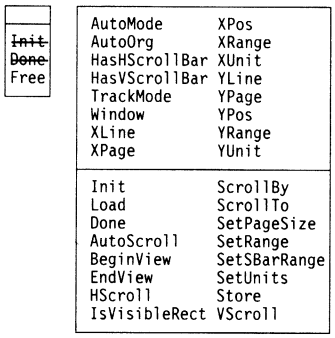
Transfers the scroll bar's range and current position to or from the memory location pointed to by *DataPtr*. If *TransferFlag* is *tf_GetData*, a record holding the range and position is transferred to the memory location. If *TransferFlag* is *tf_SetData*, the record at the memory location is retrieved and its values are used to set the range and position of the scroll bar.

Transfer returns the number of bytes stored in or retrieved from the memory location. The transfer record is defined as follows:

```
TScrollBarTransferRec = record
  LowValue: Integer;
  HighValue: Integer;
  Position: Integer;
end;
```

TScroller — OWindows

TObject TScroller



TScroller objects exist in the *Scroller* field of *TWindow* and descendant objects. The scroller object provides an automatic window-scrolling mechanism that works in conjunction with horizontal or vertical window scroll bars or without any scroll bars. It supports a technique called auto-scrolling, which does not require scroll bars.

Typically, you construct and manipulate *TScroller* objects from within the methods of their owner window objects.

Fields

AutoMode

```
AutoMode: Boolean;
```

Read/write

AutoMode is *True* if the scroller is in auto-scrolling mode. By default, *AutoMode* is *True*.

AutoOrg

```
AutoOrg: Boolean;
```

When *AutoOrg* is *True*, the origin of the display context (DC) passed to the parent window's *Paint* method is automatically adjusted so all calls using that DC reflect the position of the scroll bars. This frees you from having

to manually adjust the coordinates used when painting the window's client area. When *AutoOrg* is *False*, you must do this mapping manually.

If the scrollable range can exceed 32767, *AutoOrg* must be set to *False*.

Note that when *AutoOrg* is *True*, child windows are automatically repositioned based on the scroll bar positions. When *AutoOrg* is *False*, child windows are not supported.

HasHScrollBar `HasHScrollBar: Boolean;` **Read/write**

If the owner window has a horizontal window scroll bar, *HasHScrollBar* is *True*.

HasVScrollBar `HasVScrollBar: Boolean;` **Read/write**

If the owner window has a vertical window scroll bar, *HasVScrollBar* is *True*.

TrackMode `TrackMode: Boolean;` **Read/write**

TrackMode is *True* if the scroller automatically tracks scroll bar thumb movements by scrolling the window. By default, *TrackMode* is *True*.

Window `Window: PWindow;` **Read only**

Window points to the *TScroller*'s owner window.

XLine `XLine: Integer;` **Read/write**

XLine is the number of *XUnit* units to scroll horizontally in response to a click on the scroll bar arrow. The default value is 1.

XPage `XPage: Integer;` **Read/write**

XPage is the number of *XUnit* units to scroll horizontally in response to a click on the scroll bar's thumb area. By default, *XPage* is equal to the current width of the window in *XUnit* units. Resizing the window updates this value.

XPos `XPos: Longint;` **Read only**

XPos is the scroller's current horizontal position in terms of *XUnit* units.

XRange `XRange: Longint;` **Read only**

XRange is the total number of horizontal *XUnit* units the window can be scrolled. *XRange* values are specified in the *Init* constructor, but can be modified later.

XUnit `XUnit: Integer;` **Read only**

XUnit is the smallest number of device units (pixels) that the window can be horizontally scrolled. *XUnit* values are specified in the *Init* constructor but can be modified later through method calls.

YLine

```
YLine: Integer;
```

Read/write

YLine is the number of *YUnit* units to scroll vertically in response to a click on the scroll bar arrow. The default value is 1.

YPage

```
YPage: Integer;
```

Read/write

YPage is the number of *YUnit* units to scroll vertically in response to a click on the scroll bar's thumb area. By default, *YPage* is equal to the current height of the window in *YUnit* units. Resizing the window updates this value.

YPos

```
YPos: Longint;
```

Read only

YPos is the scroller's current vertical position in terms of *YUnit* units.

YRange

```
YRange: Longint;
```

Read only

YRange is the total number of vertical *YUnit* units the window can be scrolled. *YRange* values are specified in the *Init* constructor but can be modified later.

YUnit

```
YUnit: Integer;
```

Read only

YUnit is the smallest number of device units (pixels) that the window can be vertically scrolled. *YUnit* values are specified in the *Init* constructor but can be modified later through method calls.

Methods

Init

```
constructor Init(TheWindow: PWindow; TheXUnit, TheYUnit: Integer; TheXRange,
  TheYRange: Longint);
```

Constructs a new *TScroller* object with *TheWindow* as the owner window, and *TheXUnit*, *TheYUnit*, *TheXRange*, and *TheYRange* as *XUnit*, *YUnit*, *XRange* and *YRange*, respectively. Sets *AutoMode* and *TrackMode* to *True* and sets *HasHScrollBar* and *HasVScrollBar* depending on the scroll bar attributes of the owner window.

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a scroller from the stream *S* by first calling *TObject.Init* and then reading the *TScroller* fields, with the exception of *XPage*, *YPage*, *XPos* and *YPos*.

Done

```
destructor Done; virtual;
```

Sets the owning window's *Scroller* field to **nil**, then calls the *Done* destructor inherited from *TObject* to dispose of the scroller object.

AutoScroll

```
procedure AutoScroll; virtual;
```

Override: Sometimes

Depending on the position of the mouse cursor, performs auto-scrolling.

See also: *TWindow.WMTimer*

BeginView

```
procedure BeginView(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
```

Override: Sometimes

Sets the origin of the owner window's paint display context (*PaintDC*) according to the current scroller position.

EndView

```
procedure EndView; virtual;
```

Override: Sometimes

Updates the position of the owner window's scroll bars to be in sync with the position of the *TScroller*.

HScroll

```
procedure HScroll(ScrollRequest: Word; ThumbPos: Integer); virtual;
```

Override: Never

Responds to horizontal scroll bar events by changing the position of the scroller and the horizontal scroll bar.

See also: *TWindow.WMHScroll*

IsVisibleRect

```
function IsVisibleRect(X, Y: Longint; XExt, YExt: Integer): Boolean;
```

Override: Seldom

Returns *True* if any portion of the rectangle specified by the passed arguments is currently visible in the owner window.

ScrollBy

```
procedure ScrollBy(Dx, Dy: Longint);
```

Override: Seldom

Scrolls by the amounts specified by *Dx* and *Dy*. Also updates the window's display.

ScrollTo

```
procedure ScrollTo(X, Y: Longint);
```

Override: Sometimes

Scrolls to the position specified by *X* and *Y*. Also updates the window's display.

SetPageSize

```
procedure SetPageSize; virtual;
```

Override: Sometimes

Sets the *XPage* and *YPage* fields to be the owner window's current width and height, respectively.

See also: *TWindow.WMSize*

SetRange

```
procedure SetRange(TheXRange, TheYRange: Longint);
```

Override: Never

Overrides the *XRange* and *YRange* values passed in the call to *Init* by setting them to *TheXRange* and *TheYRange*. Then calls *SetSBarRange* to set the range of the owner window's scroll bars.

See also: *TScroller.SetSBarRange*

SetSBarRange

```
procedure SetSBarRange; virtual;
```

Override: Never

Sets the range of the of the owner window's scroll bars to be in sync with the range of the *TScroller*.

See also: *TWindow.SetupWindow*

SetUnits

```
procedure SetUnits(TheXUnit, TheYUnit: Longint);
```

Sets the *XUnit* and *YUnit* fields to *TheXUnit* and *TheYUnit*, respectively.

Store

```
procedure Store(var S: TStream);
```

Stores the scroller on the stream *S* by writing the *TScroller* fields, with the exception of *XPage*, *YPage*, *XPos* and *YPos*.

VScroll

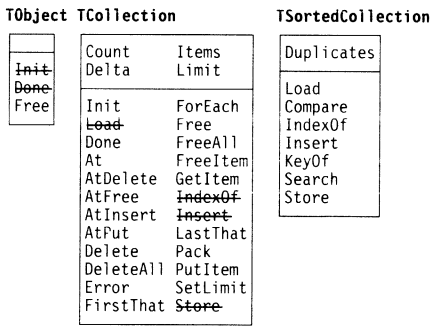
```
procedure VScroll(ScrollRequest: Word; ThumbPos: Integer); virtual;
```

Override: Never

Responds to vertical scroll bar events by changing the position of the scroller and the vertical scroll bar.

See also: *TWindow.WMVScroll*

TSortedCollection — Objects



TSortedCollection is a specialized derivative of *TCollection* implementing collections sorted by key. Sorting is implied by a virtual *Compare* method, which you override to provide your own definition of element ordering. As new items are added, they are automatically inserted in the order given by the *Compare* method. Items can be located using the binary search method, *Search*. The virtual *KeyOf* method, which returns a pointer for *Compare*, can also be overridden if *Compare* needs additional information.

TSortedCollection implements sorted collections both with or without duplicate keys. The *Duplicates* field controls whether duplicates are allowed. It defaults to *False*, indicating that duplicate keys are not allowed, but after creating a sorted collection, you can set *Duplicates* to *True* to allow elements with duplicate keys in the collection.

Field

Duplicates

```
Duplicates: Boolean;
```

Read/write

Determines whether the collection accepts items with duplicate keys. By default, *Duplicates* is *False*, and calling *Insert* for an item whose key matches that of an item already in the collection causes the collection to not insert the new item. The collection keeps only the first item inserted with a given key.

If you set *Duplicates* to *True*, the collection inserts duplicate-key items immediately before the first existing item with the same key.

Methods

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a sorted collection from the stream *S* by first calling *TCollection.Load* and then reading the *TSortedCollection* field *Duplicates*.

See also: *TCollection.Load*

Compare

```
function Compare(Key1, Key2: Pointer): Integer; virtual;
```

Override: Always

Compare is an abstract method that must be overridden in all descendant types. *Compare* should compare the two key values and return a result as follows: –1 if *Key1* < *Key2*; 0 if *Key1* = *Key2*; and 1 if *Key1* > *Key2*.

Key1 and *Key2* are pointer values, as extracted from their corresponding collection items by the *TSortedCollection.KeyOf* method. The *TSortedCollection.Search* method implements a binary search through the collection's items using *Compare* to compare the items.

See also: *TSortedCollection.KeyOf*, *TSortedCollection.Compare*

IndexOf

```
function IndexOf(Item: Pointer): Integer; virtual;
```

Override: Never

Uses *TSortedCollection.Search* to find the index of the given *Item*. If the item is not in the collection, *IndexOf* returns –1. The actual implementation of *TSortedCollection.IndexOf* is:

```
if Search(KeyOf(Item), I) then IndexOf := I else IndexOf := -1;
```

See also: *TSortedCollection.Search*

Insert

```
procedure Insert(Item: Pointer); virtual;
```

Override: Never

If the target item is not found in the sorted collection, it is inserted at the correct index position. Calls *TSortedCollection.Search* to determine if the item exists, and if not, where to insert it. The actual implementation of *TSortedCollection.Insert* is:

```
if not Search(KeyOf(Item), I) or Duplicates then AtInsert(I, Item);
```

See also: *TSortedCollection.Search*

KeyOf

```
function KeyOf(Item: Pointer): Pointer; virtual;
```

Override: Sometimes

Given an *Item* from the collection, *KeyOf* should return the corresponding key of the item. The default *TSortedCollection.KeyOf* simply returns *Item*. *KeyOf* is overridden in cases where the key of the item is not the item itself.

See also: *TSortedCollection.IndexOf*

Search

```
function Search(Key: Pointer; var Index: Integer): Boolean; virtual;
```

Override: Seldom

Returns *True* if the item identified by *Key* is found in the sorted collection. If the item is found, *Index* is set to the found index; otherwise, *Index* is set to the index where the item would be placed if inserted.

See also: *TSortedCollection.Compare*, *TSortedCollection.Insert*

Store

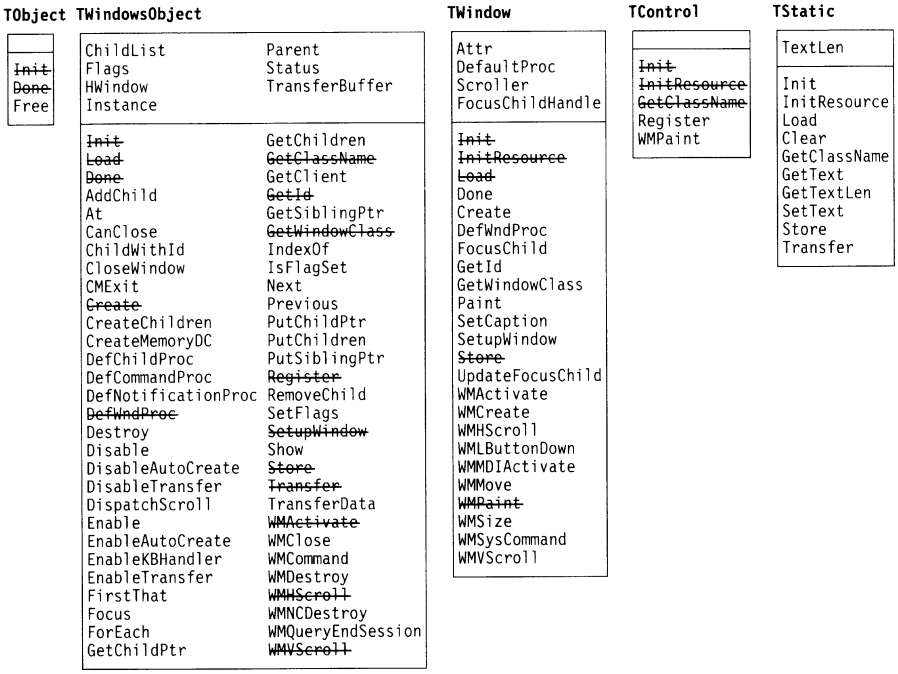
```
procedure Store(var S: TStream);
```

Stores the sorted collection and all its items on the stream *S* by calling *TCollection.Store* to write the collection and then writing the *Duplicates* field to the stream.

See also: *TCollection.Store*

TStatic

ODialogs



TStatic is an interface object that represents a corresponding static control element in Windows.

Field

TextLen

```
TextLen: Word;
```

Read only

TextLen holds the size of the text buffer for static controls. The number of characters that can actually be stored in the static control is one less than *TextLen*, because of the null terminator on the string. *TextLen* is also the number of bytes transferred by the *Transfer* method.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; ATitle: PChar;
  X, Y, W, H: Integer, ATextLen: Word);
```

Override: Seldom

Constructs a static control object with the passed parent window (*AParent*), control ID (*AnID*), text (*ATitle*), position relative to the origin of the parent window's client area (*X*, *Y*), width (*W*), height (*H*), and text

length(*ATextLen*). The static control is left-justified because *TStatic.Init* adds the Windows style *ss_Left* to the object's *Attr.Style* field. It also removes the *ws_TabStop* style. *Init* then calls *DisableTransfer* to exclude *TStatic* objects from the transfer mechanism, by default.

InitResource

```
constructor InitResource(AParent: PWindowsObject; ResourceID,
  ATextLen: Word);
```

Associates a *TStatic* object with the static-control resource specified by *ResourceID* by calling the *InitResource* constructor inherited from *TControl*. Sets the *TextLen* field to *ATextLen*.

See also: *TControl.InitResource*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a static control from the stream *S* by calling the *Load* constructor inherited from *TControl* and then reading the *TextLen* field.

See also: *TControl.Load*

Clear

```
procedure Clear; virtual;
```

Override: Seldom

Clears the static control's text.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Seldom

Returns the name of *TStatic*'s window class, 'Static'.

GetText

```
function GetText(ATextString: PChar; MaxChars: Integer): Integer; virtual;
```

Override: Seldom

Retrieves the static control's text and stores it in the *ATextString* argument. *MaxChars* specifies the maximum size of *ATextString*. *GetText* returns the size of the retrieved string.

GetTextLen

```
function GetTextLen: Integer;
```

Returns the length of the static control's text string.

SetText

```
procedure SetText(ATextString: PChar); virtual;
```

Override: Seldom

Sets the static control's text to be the string passed in *ATextString*.

Store

```
procedure Store(var S: TStream);
```

Stores the static control on the stream *S* by calling the *Store* method inherited from *TControl* and then writing the *TextLen* field.

See also: *TControl.Store*

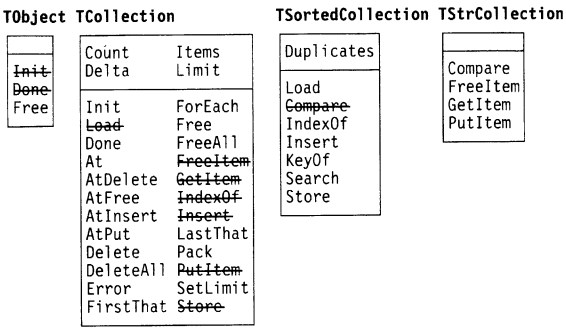
Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
```

Override: Sometimes

Transfers *TextLen* characters of the current text of the static control to or from the memory location pointed to by *DataPtr*. If *TransferFlag* is *tf_GetData*, the text is transferred to the memory location. If *TransferFlag* is *tf_SetData*, the static control's text is set to the text at the memory location. Transfer returns *TextLen*, the number of bytes stored in or retrieved from the memory location. If *TransferFlag* is *tf_SizeData*, *Transfer* returns the size of the transfer data.

TStrCollection — Objects



TStrCollection is a simple derivative of *TSortedCollection* implementing a sorted list of ASCII strings. The *TStrCollection.Compare* method is overridden to provide the conventional lexicographic ASCII string ordering. You can override *Compare* to allow for other orderings, such as those for non-English character sets.

Methods

Compare

```
function Compare(Key1, Key2: Pointer): Integer; virtual;
```

Override: Sometimes

Compares the strings *Key1*^ and *Key2*^ as follows: return –1 if *Key1* < *Key2*; 0 if *Key1* = *Key2*; and +1 if *Key1* > *Key2*.

See also: *TSortedCollection.Search*

FreeItem

```
procedure FreeItem(Item: Pointer); virtual;
```

Override: Seldom

Removes the string *Item*^ from the sorted collection and disposes of the string.

GetItem

```
function GetItem(var S: TStream): Pointer; virtual;
```

Override: Seldom

By default, reads a string from the stream by calling *S.ReadStr*.

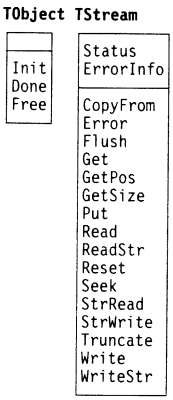
See also: *TStream.ReadStr*

PutItem `procedure PutItem(var S: TStream; Item: Pointer); virtual;`

Override: Seldom By default, writes the string *Item^* to the stream by calling *S.WriteStr*.

See also: *TStream.WriteStr*

TStream Objects



TStream is a general abstract object providing polymorphic I/O to and from a storage device. You can create your own derived stream objects by overriding the virtual methods: *GetPos*, *GetSize*, *Read*, *Seek*, *Truncate*, and *Write*. ObjectWindows itself does this to derive *TDosStream* and *TEmsStream*. For buffered derived streams, you must also override *TStream.Flush*.

Fields

Status `Status: Integer` **Read/write**

Status indicates the current status of the stream using one of the *stXXXX* constants *stOk*, *stError*, *stInitError*, *stReadError*, *stWriteError*, *stGetError*, or *stPutError*.

If *Status* is not *stOk* all operations on the stream are suspended until *Reset* is called.

See also: *stXXXX* constants

ErrorInfo `ErrorInfo: Integer` **Read/write**

ErrorInfo contains additional information when *Status* is not *stOk*. For *Status* values of *stError*, *stInitError*, *stReadError*, and *stWriteError*, *ErrorInfo* contains the DOS or EMS error code, if one is available. When *Status* is *stGetError*, *ErrorInfo* contains the object type ID (the *ObjType* field of a *TStreamRec*) of the unregistered object type. When *Status* is *stPutError*, *ErrorInfo* contains the VMT data segment offset (the *VmtLink* field of a *TStreamRec*) of the unregistered object type.

Methods

CopyFrom

```
procedure CopyFrom(var S: TStream; Count: Longint);
```

Copies *Count* bytes from stream *S* to the stream object. For example:

```
{Create a copy of entire stream}
NewStream := New(TEmsStream, Init(OldStream^.GetSize));
OldStream^.Seek(0);
NewStream^.CopyFrom(OldStream, OldStream^.GetSize);
```

See also: *TStream.GetSize*, *TObject.Init*

Error

```
procedure Error(Code, Info: Integer); virtual;
```

Override: Sometimes

Called whenever a stream error occurs. The default *TStream.Error* stores *Code* and *Info* in the *Status* and *ErrorInfo* fields and then, if the global variable *StreamError* is not **nil**, calls the procedure pointed to by *StreamError*. Once an error has occurred, all stream operations on the stream are suspended until *Reset* is called.

See also: *TStream.Reset*, *StreamError* variable

Flush

```
procedure Flush; virtual;
```

Override: Sometimes

Flushes any buffers by clearing the read buffer, by writing the write buffer, or both. The default *TStream.Flush* does nothing and must be overridden in descendant types that implement buffers.

See also: *TBufStream.Flush*

Get

```
function Get: PObject;
```

Reads an object from the stream. The object must have been previously written to the stream by *TStream.Put*. *Get* first reads an object type ID (a word) from the stream. It then finds the corresponding object type by comparing the ID to the *ObjType* field of all registered object types (see the *TStreamRec* type), and finally calls the *Load* constructor of that object type to create and load the object. If the object type ID read from the stream is zero, *Get* returns a **nil** pointer; if the object type ID has not been registered

(using *RegisterType*), *Get* calls *TStream.Error* and returns a **nil** pointer; otherwise, *Get* returns a pointer to the newly created object.

See also: *TStream.Put, RegisterType, TStreamRec, Load* methods

GetPos

```
function GetPos: Longint; virtual;
```

Override: Always

Returns the value of the stream's current position. This is an abstract method that must be overridden.

See also: *TStream.Seek*

GetSize

```
function GetSize: Longint; virtual;
```

Override: Always

Returns the total size of the stream. This is an abstract method that must be overridden.

Put

```
procedure Put(P: PObject);
```

Writes an object to the stream. The object can later be read from the stream using *TStream.Get*. *Put* first finds the type registration record of the object by comparing the object's VMT offset to the *VmtLink* field of all registered object types (see the *TStreamRec* type). It then writes the object type ID (the *ObjType* field of the registration record) to the stream, and finally calls the *Store* method of that object type to write the object. If the *P* argument passed to *Put* is **nil**, *Put* writes a word containing zero to the stream. If the object type of *P* has not been registered (using *RegisterType*), *Put* calls *TStream.Error* and doesn't write anything to the stream.

See also: *TStream.Get, RegisterType, TStreamRec, Store* methods

Read

```
procedure Read(var Buf; Count: Word); virtual;
```

Override: Always

Reads *Count* bytes from the stream into *Buf* and advances the current position of the stream by *Count* bytes. If an error occurs, *Read* calls *Error*, and fills *Buf* with *Count* bytes of zero. This is an abstract method that must be overridden in all descendant types.

See also: *TStream.Write, TStream.Error.*

ReadStr

```
function ReadStr: PString;
```

Reads a string from the current position of the stream, returning a *PString* pointer. *TStream.ReadStr* calls *GetMem* to allocate (Length+1) bytes for the string.

See also: *TStream.WriteStr*

Reset

```
procedure Reset;
```

Resets any stream error condition by setting *Status* and *ErrorInfo* to zero. This method lets you continue stream processing following an error condition that you have corrected.

See also: *TStream.Status*, *TStream.ErrorInfo*, *stXXXX* error codes

Seek

```
procedure Seek(Pos: Longint); virtual;
```

Override: Always

Sets the current position to *Pos* bytes from the start of the stream. The start of a stream is position 0. This is an abstract method that must be overridden by all descendants.

See also: *TStream.GetPos*

StrRead

```
function StrRead: PChar;
```

Reads a null-terminated string from the stream by first reading the length of the string and then reading that number of characters. Returns a pointer to the null-terminated string read.

See also: *TStream.StrWrite*

StrWrite

```
procedure StrWrite(P: PChar);
```

Writes the null-terminated string *P* to the stream by first writing the length of the string and then writing that number of characters.

See also: *TStream.StrRead*

Truncate

```
procedure Truncate; virtual;
```

Override: Always

Deletes all data on the stream from the current position to the end. This is an abstract method that must be overridden by all descendants.

See also: *TStream.GetPos*, *TStream.Seek*

Write

```
procedure Write(var Buf; Count: Word); virtual;
```

Override: Always

Writes *Count* bytes from *Buf* onto the stream and advances the current position of the stream by *Count* bytes. If an error occurs, *Write* calls *Error*. This is an abstract method that must be overridden in all descendant types.

See also: *TStream.Read*, *TStream.Error*.

WriteStr

```
procedure WriteStr(P: PString);
```

Writes the string *P*^ to the stream, starting at the current position.

See also: *TStream.ReadStr*

TStreamRec type

Declaration

```
TStreamRec = record
  ObjType: Word;
  VmtLink: Word;
  Load: Pointer;
  Store: Pointer;
  Next: Word;
end;
```

Function

An ObjectWindows object type must have a registered *TStreamRec* before its objects can be loaded or stored on a *TStream* object. The *RegisterTypes* routine registers an object type by setting up a *TStreamRec* record.

The fields in the stream registration record are defined as follows:

Table 21.27
Stream record fields

Field	Contents
ObjType	A unique numerical id for the object type
VmtLink	A link to the object type's virtual method table entry
Load	A pointer to the object type's *Load* constructor
Store	A pointer to the object type's *Store* method
Next	A pointer to the next *TStreamRec*

ObjectWindows reserves object type IDs (*ObjType*) values 0 through 999 for its own use. Programmers can define their own values in the range 1,000 to 65,535.

By convention, a *TStreamRec* for a *Txxxx* object type is called *Rxxxx*. For example, the *TStreamRec* for a *TCalculator* type is called *RCalculator*, as shown in the following code:

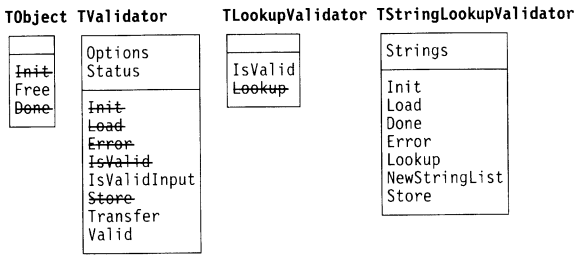
```
type
  TCalculator = object(TDialog)
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
    :
  end;

const
  RCalculator: TStreamRec = (
    ObjType: 2099;
    VmtLink: Ofs(TypeOf(TCalculator)^);
    Load: @TCalculator.Load;
    Store: @TCalculator.Store);
```

```
begin
  RegisterType(RCalculator);
   :
end;
```

See also *RegisterType*

TStringLookupValidator — Validate



A string-lookup validator object verifies the data in its associated input line by searching through a collection of valid strings. Use string-lookup validators when your input line needs to accept only members of a certain set of strings.

Field

Strings

```
Strings: PStringCollection;
```

Points to a string collection containing all the valid strings the user can type. If *Strings* is **nil**, all input will be invalid.

Methods

Init

```
constructor Init(AStrings: PStringCollection);
```

Constructs a string-lookup validator object by first calling the *Init* constructor inherited from *TLookupValidator* and then setting *Strings* to *AStrings*.

See also: *TLookupValidator.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a string-lookup validator object from the stream *S* by first calling the *Load* constructor inherited from *TLookupValidator* and then reading the string collection *Strings*.

See also: *TLookupValidator.Load*

Done `destructor Done; virtual;`

Disposes of the list of valid strings by calling *NewStringList(**nil**)* and then disposes of the string-lookup validator object by calling the *Done* destructor inherited from *TLookupValidator*.

See also: *TLookupValidator.Done, TStringLookupValidator.NewStringList*

Error `procedure Error; virtual;`

Displays a message box indicating that the typed string does not match an entry in the string list.

Lookup `function Lookup(const S: string): Boolean; virtual;`

Returns *True* if the string passed in *S* matches any of the strings in *Strings*. Uses the *Search* method of the string collection to determine if *S* is present.

See also: *TSortedCollection.Search*

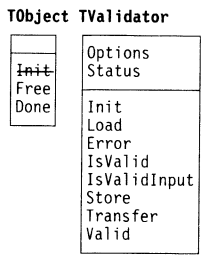
NewStringList `procedure NewStringList(AStrings: PStringCollection);`

Sets the list of valid input strings for the string-lookup validator. Disposes of any existing string list and then sets *Strings* to *AStrings*. Passing **nil** in *AStrings* disposes of the existing list without assigning a new one.

Store `procedure Store(var S: TStream);`

Stores the string-lookup validator on the stream *S* by first calling the *Store* method inherited from *TValidator* and then writing the string collection held in *Strings*.

TValidator — Validate



TValidator defines an abstract data validation object. You will never actually create an instance of *TValidator*, but it provides the abstract functions for the other data validation objects.

Fields

Options

```
Options: Word;
```

Options is a bitmapped field used to control options for various descendants of *TValidator*. By default, *TValidator.Init* clears all the bits in *Options*.

See also: *voXXXX* constants

Status

```
Status: Word;
```

Status indicates the status of the validator object. If *Status* is *vsOK*, the validator object constructed correctly. Any value other than *vsOK* indicates that an error occurred.

See also: *TInputLine.Valid*, *ValidatorOK* constant

Methods

Init

```
constructor Init;
```

Constructs an abstract validator object by first calling the *Init* constructor inherited from *TObject* and then setting the *Options* and *Status* fields to zero.

See also: *TObject.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs a validator object by calling the *Init* constructor inherited from *TObject* and then reads the *Options* word from the stream *S*.

See also: *TObject.Init*

Error

```
procedure Error; virtual;
```

Error is an abstract method called by *Valid* when it detects that the user has entered invalid information. By default, *TValidator.Error* does nothing, but descendant types can override *Error* to provide feedback to the user.

IsValid

```
function IsValid(const S: string): Boolean; virtual;
```

By default, *TValidator.IsValid* returns *True*. Descendant validator types can override *IsValid* to validate data for a completed input line. If an input line has an associated validator object, its *Valid* method calls the validator object's *Valid* method, which in turn calls *IsValid* to determine whether the contents of the input line are valid.

See also: *TInputLine.Valid*, *TValidator.Valid*

IsValidInput

```
function IsValidInput(var S: string; SuppressFill: Boolean): Boolean;
  virtual;
```

If an input line has an associated validator object, it calls *IsValidInput* after processing each keyboard event. This gives validators such as filter validators an opportunity to catch errors before the user fills the entire item or screen.

By default, *TValidator.IsValidInput* returns *True*. Descendant data validators can override *IsValidInput* to validate data as the user types it, returning *True* if *S* holds valid data and *False* otherwise.

S is the current input string. *SuppressFill* determines whether the validator should automatically format the string before validating it. If *SuppressFill* is *True*, validation takes place on the unmodified string *S*. If *SuppressFill* is *False*, the validator should apply any filling or padding before validating data. Of the standard validator objects, only *TPXPictureValidator* checks *SuppressFill*.

Because *S* is a **var** parameter, *IsValidInput* can modify the contents of the input string, such as forcing characters to uppercase or inserting literal characters from a format picture. *IsValidInput* should not, however, delete invalid characters from the string. By returning *False*, *IsValidInput* indicates that the input line should erase the offending characters.

Store

```
procedure Store(var S: TStream);
```

Writes the validator object to the stream *S* by writing the value of the *Options* field.

Transfer

```
function Transfer(var S: String; Buffer: Pointer; Flag: TVTransfer): Word;
  virtual;
```

Allows a validator to take over setting and reading the values of its associated input line, which is mostly useful for validators that check non-string data, such as numeric values. For example, *TRangeValidator* uses *Transfer* to read and write *Longint*-type values to a data record, rather than transferring an entire string.

By default, input lines with validators give the validator the first chance to respond to *DataSize*, *GetData*, and *SetData* by calling the validator's *Transfer* method. If *Transfer* returns anything other than 0, it indicates to the input line that it has handled the appropriate transfer. The default action of *TValidator.Transfer* is to return 0 always. If you want the validator to transfer data, you need to override its *Transfer* method.

Transfer's first two parameters are the associated input line's text string and the *GetData* or *SetData* data record. Depending on the value of *Flag*, *Transfer* can set *S* from *Buffer* or read the data from *S* into *Buffer*. The return value is always the number of bytes transferred.

If *Flag* is *vtDataSize*, *Transfer* doesn't change either *S* or *Buffer* but just returns the data size. If *Flag* is *vtSetData*, *Transfer* reads the appropriate number of bytes from *Buffer*, converts them into the proper string form, and sets them into *S*, returning the number of bytes read. If *Flag* is *vtGetData*, *Transfer* converts *S* into the appropriate data type and writes the value into *Buffer*, returning the number of bytes written.

See also: *TInputLine.DataSize*, *TInputLine.GetData*, *TInputLine.SetData*

Valid

```
function Valid(const S: string): Boolean;
```

Returns *True* if *IsValid(S)* returns *True*. Otherwise, calls *Error* and returns *False*. A validator's *Valid* method is called by the *Valid* method of its associated input line.

Input lines with associated validator objects call the validator's *Valid* method under two conditions: either the input line has its *ofValidate* option set, in which case it calls *Valid* when it loses focus, or the dialog box that contains the input line calls *Valid* for all its controls, usually because the user requested to close the dialog box or accept an entry screen.

See also: *TInputLine.Valid*, *TValidator.Error*, *TValidator.IsValid*

TVTransfer type — Validate

Declaration

```
TVTransfer = (vtDataSize, vtSetData, vtGetData);
```

Function Validator objects use parameters of type *TVTransfer* in their *Transfer* methods to control data transfer when setting or reading the value of the associated input line.

See also *TValidator.Transfer*

TWndClass type — WinTypes

Declaration

```
TWndClass = record
  style: Word;
  lpfnWndProc: TFarProc;
  cbClsExtra: Integer;
```

```
    cbWndExtra: Integer;
    hInstance: THandle;
    hIcon: HIcon;
    hCursor: HCursor;
    hbrBackground: HBrush;
    lpszMenuName: PChar;
    lpszClassName: PChar;
  end;
```

Function The *TWndClass* record holds the attributes of a window class, also known as the registration attributes, as they are registered with the *RegisterClass* function.

The *style* field holds the class style. It can hold one or a combination of the *cs_* class style constants.

The *lpfnWndProc* field points to the window's window function, the routine that receives and processes messages.

cbClsExtra is the number of bytes to be allocated at the end of the *TWndClass* record. These are called the class' extra bytes, and can be accessed with the *GetWindowLong* or *GetWindowWord* functions, or set with the *SetWindowLong* or *SetWindowWord* functions.

cbWndExtra gives the number of bytes to allocate at the end of the window instance.

hInstance is an instance handle that must indicate the class module. It must not be zero.

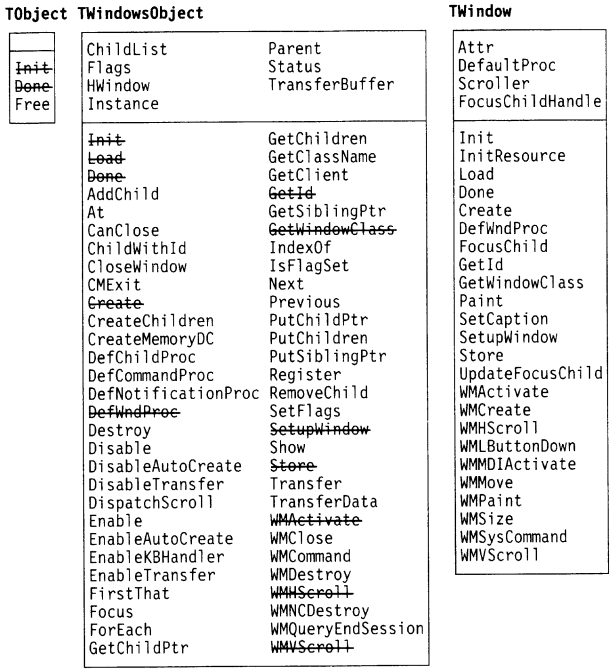
The *hIcon*, *hCursor*, and *hbrBackground* fields are the handles of the class' icon and cursor, and the class' background color, respectively. The background color should be a color value (one of the standard system colors, given by a *color_* constant, incremented by one) or the handle of a brush for painting the background. If *hbrBackground* is zero, the application's background must be painted whenever its client area is painted. The need for this can be determined by processing the *wm_EraseBkgnd* message or by checking the *fErase* field of the *TPaintStruct* record created by *BeginPaint*.

The *lpszMenuName* and *lpszClassName* field both point to null-terminated strings, being the resource name of the class menu and the name of the class, respectively.

See also *TWindowsObject.GetWindowClass*

TWindow

OWindows



TWindow defines the fundamental behavior for all window and control objects. Object instances of *TWindow* are empty generic windows, but they can define menus, cursors and icons.

Fields

Attr

```
Attr: TWindowAttr;
```

Read/write

Attr holds a *TWindowAttr* record, which defines the windows creation attributes, characteristics that influence the creation of the window object's corresponding interface element. These include the window's associated text, style, extended style, position and size, window handle, and control ID. These attributes are set to defaults in the object's *Init* constructor but can be overridden in a descendant type's constructor.

See also: *TWindowAttr* type

DefaultProc

```
DefaultProc: TFarProc;
```

Read only

DefaultProc holds the address of the default window procedure, which defines the Windows default processing for Windows messages.

FocusChildHandle `FocusChildHandle: THandle;` **Read only**

FocusChildHandle stores the handle to the window's child window that had the focus the last time the window was activated. Windows doesn't automatically keep track of focused child windows, so when you reactivate a window or restore it from an icon, ObjectWindows makes sure focus is restored to the child window that last had focus.

You can manipulate *FocusChildHandle* using the methods *FocusChild* and *UpdateFocusChild*.

See also: *TWindow.FocusChild, TWindow.UpdateFocusChild*

Scroller `Scroller: PScroller;` **Read/write**

Scroller holds a pointer to a *TScroller* object, which is the window's scroller, if any. A scroller should be constructed in the window's *Init* constructor.

Methods

Init `constructor Init(AParent: PWindowsObject; ATitle: PChar);`

Override: Often

Constructs a window object with the parent window passed in *AParent* and the associated text (caption for windows) passed in *ATitle*. *AParent* should be **nil** for main windows, which have no parent. The object's *Attr.Style* field is set to *ws_OverlappedWindow* unless the window is an MDI child window. In that case, the *Attr.Style* field is set to *ws_ClipSiblings*. The position and extent fields in the *Attr* record are set to defaults appropriate for overlapped and pop-up windows.

You can override *Init* in your descendant types as long as you explicitly call *TWindow.Init*. Then you can reset the *Attr* fields. *Scroller*, by default, is set to **nil**.

InitResource `constructor InitResource(AParent: PWindowsObject; ResourceID: Word);`

Constructs an interface object to be associated with a screen element (usually a control) based on a resource definition. Calls *TWindowsObject.Init* to construct the object.

See also: *TWindowsObject.Init*

Load `constructor Load(var S: TStream);`

Constructs and loads a window from the stream *S* by first calling *TWindowsObject.Load* and then reading and getting the additional fields (*Attr* and *Scroller*) introduced by *TWindow*.

See also: *TWindowsObject.Load*

Done

```
destructor Done; virtual;
```

Override: Often

Disposes of the *TScroller* object in *Scroller*, if any, before calling the *Done* destructor inherited from *TWindowsObject* to dispose of the entire object.

Create

```
function Create: Boolean; virtual;
```

Creates the window object's corresponding screen element unless the object was constructed with *InitResource*, in which case the interface element already exists. *Create* calls *Register* to register the window class if not already registered. *Create* then creates the window and calls *SetupWindow*, which you can define to initialize the newly created window, usually by creating child windows and drawing graphics or text. *Create* returns *True* if successful. If unsuccessful, it returns *False* and calls *Error*.

Normally, you don't call *Create* directly. *Create* is called by *TApplication.MakeWindow*, which first checks to make sure memory is available.

See also: *TWindowsObject.Register, TWindowsObject.SetupWindow, TWindow.InitResource, TWindowsObject.Error, TApplication.MakeWindow*

DefWndProc

```
procedure DefWndProc(var Msg: TMessage); virtual;
```

Override: Never

Calls the window's default procedure which handles default processing for incoming Windows messages. It stores the result of this call in the *Result* field of the message record, *Msg*.

FocusChild

```
procedure FocusChild;
```

Called by *WMActivate* and *WMSysCommand* to set the input focus to the child window with the handle in *FocusChildHandle*.

See also: *TWindow.WMActivate, TWindow.WMSysCommand*

GetID

```
function GetId: Integer; virtual;
```

Override: Seldom

Returns the window identifier, such as a control ID.

GetWindowClass

```
procedure GetWindowClass(var AWndClass: TWndClass); virtual;
```

Override: Often

Fills a window class record, passed in *AWndClass*, with default values for its registration attributes. The style field is set to *cs_HRedraw* or *cs_VRedraw*. The icon is set to a generic icon and the cursor is set to the stock arrow cursor. The background color is set to the system's window background color. The name of the class to be registered is retrieved through a call to *GetClassName*.

See also: *TWindowsObject.GetClassName*, *TWindowsObject.Register*, *TWindow.Create*

Paint

```
procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
```

Override: Often

Serves as a placeholder for descendant types that define *Paint* methods. *Paint* is called in response to a request (*wm_Paint*) from Windows to redisplay the window's contents. Use *PaintDC* as the display context. It is already obtained before the call to Paint and released after *Paint*. The *PaintInfo* record contains information about the paint request.

See also: *TWindow.WMPaint*

SetCaption

```
procedure SetCaption(ATitle: PChar);
```

Disposes of the text in the window's *Attr.Title* by calling *StrDispose*, then calls *StrNew* to allocate a copy of the string in *ATitle* to *Attr.Title*. Calls the *SetWindowText* API function to update the window title.

SetupWindow

```
procedure SetupWindow; virtual;
```

Override: Often

Sets up the newly created window. If the window is an MDI child window, *SetupWindow* calls *SetFocus* to give the new window the focus. If the window has a scroller object, *SetupWindow* calls *SetSBarRange* to set the range of its scroll bars.

See also: *TScroller.SetSBarRange*

Store

```
procedure Store(var S: TStream);
```

Stores the window on the stream *S* by first calling *TWindowsObject.Store* and then writing and putting the additional fields (*Attr* and *Scroller*) introduced by *TWindow*.

See also: *TWindowsObject.Store*

UpdateFocusChild

```
procedure UpdateFocusChild;
```

Sets *FocusChildHandle* to the handle of the child window that currently has the input focus.

See also: *TWindow.FocusChildHandle*

WMActivate

```
procedure WMActivate(var Msg: TMessage); virtual wm_First + wm_Activate;
```

Override: Sometimes

For windows that intercept keyboard messages for their controls, responds to the window's losing and receiving the focus by saving the handle of the child control that currently has the focus in *FocusChildHandle* and restoring the focus.

See also: *TWindowsObject.EnableKBHandler*

WMCreate

```
procedure WMCreate(var Msg: TMessage); virtual wm_First + wm_Create;
```

Responds to the *wm_Create* message by calling *SetupWindow* and then calling *DefWndProc*. Since window creation is handled differently under ObjectWindows than under Windows, the *wm_Create* message needs to be trapped and used to set up window object attributes.

See also: *TWindow.SetupWindow*

WMHScroll

```
procedure WMHScroll(var Msg: TMessage); virtual wm_First + wm_HScroll;
```

Sometimes

For windows with scrollers, responds to horizontal window scroll bar events by calling the scroller's *HScroll* method and *DefWndProc*.

See also: *TScroller.HScroll*

WMLButtonDown

```
procedure WMLButtonDown(var Msg: TMessage); virtual wm_First + wm_LButtonDown;
```

Override: Sometimes

For windows with auto-scrolling scrollers, responds to a left mouse button click by capturing all future mouse input until the mouse button is released. If you plan to override this method to process mouse clicks, but still plan to use auto-scrolling, be sure to call this method from your *WMLButtonDown* method.

WMMDIActivate

```
procedure TWindow.WMMDIActivate(var Msg: TMessage); virtual wm_First +
  wm_MDIActivate;
```

Handles MDI Window activation by calling *WMActivate*.

See also: *TWindow.WMActivate*

WMMove

```
procedure WMMove(var Msg: TMsg); virtual wm_First + wm_Move;
```

Updates *Attr.X* and *Attr.Y* when the *wm_Move* message is received, unless the window is iconic or zoomed, in which case the message is ignored.

WMPaint

```
procedure WMPaint(var Msg: TMessage); virtual wm_First + wm_Paint;
```

Override: Seldom

Responds to the Windows *wm_Paint* message by calling the window object's *Paint* method. If the window has a scroller, *WMPaint* calls *BeginView* before calling *Paint* and *EndView* after calling *Paint*.

See also: *TWindow.Paint*, *TScroller.EndView*, *TScroller.BeginView*

WMSize

```
procedure WMSize(var Msg: TMessage); virtual wm_First + wm_Size;
```

Override: Sometimes

For windows with scrollers, responds to a window-sizing event by calling *SetPageSize* to adjust for the new window size.

See also: *TScroller.SetPageSize*

WMSysCommand

```
procedure TWindow.WMSysCommand(var Msg: TMessage); virtual wm_First +
  wm_SysCommand;
```

If the window handles keyboard input, checks the *wParam* field of *Msg* for two values before calling *DefWndProc* to handle normal processing. If *Msg.wParam* is *sc_Minimize*, meaning the window will shrink to an icon, *WMSysCommand* calls *UpdateFocusChild* before iconizing. If *Msg.wParam* is *sc_Restore*, *WMSysCommand* calls *FocusChild* to restore the input focus before restoring the window.

See also: *TWindow.FocusChild*, *TWindow.UpdateFocusChild*

WMVScroll

```
procedure WMVScroll(var Msg: TMessage); virtual wm_First + wm_VScroll;
```

Override: Sometimes

For windows with scrollers, responds to vertical window scroll bar events by calling the scroller's *VScroll* method and *DefWndProc*.

See also: *TScroller.VScroll*

TWindowAttr type — OWindows

Declaration

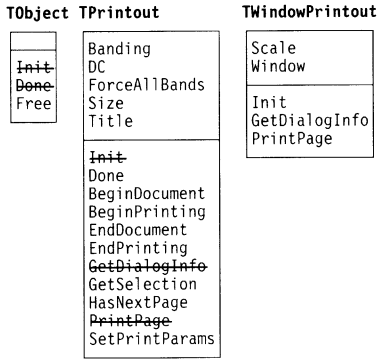
```
TWindowAttr = record
  Title: PChar;
  Style: Longint;
  ExStyle: Longint;
  X, Y, W, H: Integer;
  Param: Pointer;
  case Integer of
    0: (Menu: HMenu);                    { window menu's handle or... }
    1: (Id: Integer);                    { control's child identifier }
  end;
```

Function *TWindow* objects define their attributes in *TWindowAttr* records.

See also *TWindow.Attr*

TWindowPrintout

OPrinter



TWindowPrintout provides a printout object streamlined to print the contents of a window.

Fields

Scale

```
Scale: Boolean;
```

True if the printout should scale the image to fill the page. By default, *TWindowPrintout.Init* sets *Scale* to *True*.

Window

```
Window: PWindow;
```

Window points to the window to print.

Methods

Init

```
constructor Init(ATitle: PChar; AWindow: PWindow);
```

Constructs a window printout object by first calling the *Init* constructor inherited from *TPrintout*, passing *ATitle* and then sets *Window* to *AWindow* and sets *Scale* to *True*.

See also: *TPrintout.Init*

GetDialogInfo

```
function GetDialogInfo(var Pages: Integer): Boolean; virtual;
```

Sets *Pages* to zero and returns *False*, since the window generates only a single page of printout. This prevents the print dialog from appearing.

PrintPage

```
procedure PrintPage(Page: Word; var Rect: TRect; Flags: Word); virtual;
```

Scales the device context to the window so the printout will look right and then calls the window's *Paint* method to put the window's image on the device context.

TWindowsObject OWindows

TObject TWindowsObject

~~Init~~ ~~Done~~ Free	ChildList Parent Flags Status HWindow TransferBuffer Instance
	Init GetChildren Load GetClassName Done GetClient AddChild GetId At GetSiblingPtr CanClose GetWindowClass ChildWithId IndexOf CloseWindow IsFlagSet CMExit Next Create Previous CreateChildren PutChildPtr CreateMemoryDC PutChildren DefChildProc PutSiblingPtr DefCommandProc Register DefNotificationProc RemoveChild DefWndProc SetFlags Destroy SetupWindow Disable Show DisableAutoCreate Store DisableTransfer Transfer DispatchScroll TransferData Enable WMActivate EnableAutoCreate WMClose EnableKBHandler WMCommand EnableTransfer WMDestroy FirstThat WMHScroll Focus WMNCDestroy ForEach WMQueryEndSession GetChildPtr WMVScroll

TWindowsObject defines the fundamental behavior for all interface objects, including windows, dialog boxes, and controls. *TWindowsObject* is an abstract object type and its methods are useful only to descendant types. Its methods implement the fundamental screen-element creation and destruction behavior, window-class registration behavior, and automatic message-response mechanism.

Fields

ChildList

```
ChildList: PWindowsObject;
```

Read only

ChildList is a linked list of all of the interface object's child-window objects, such as pop-up windows, dialog boxes, and controls. *ChildList* always points to the object most recently added.

Flags

```
Flags: Byte;
```

Read/write

Flags is a byte of data whose bits are used to store the following window attributes: keyboard handling, auto-creation, transfer, MDI status, and resource creation. *Flags* contains one or more of the *wb_* constants documented in this chapter.

See also: *TWindowsObject.SetFlags*, *TWindowsObject.IsFlagSet*

HWindow

```
HWindow: HWnd;
```

Read only

HWindow holds a handle to the interface object's associated interface element. If there is no associated element, *HWindow* is equal to zero, the value of an invalid handle. Upon creation of the associated interface element (*Create*), *HWindow* is set to a new handle. Upon destruction of the associated interface element (*WMNCDestroy*), *HWindow* is set to zero.

Instance

```
Instance: TFarProc;
```

Read only

Instance points to the code executed prior to entering the shared window procedure of ObjectWindows.

Parent

```
Parent: PWindowsObject;
```

Read only

PWindowsObject points to the interface object that serves as this interface object's parent window. For example, the *Parent* field of a control object that appears in a window object would point to the window object, its parent.

Status

```
Status: Integer;
```

Read/Write

Status indicates the success of the current effort to initialize an interface object and its associated interface element. The process is successful so far if *Status* is greater than or equal to zero and unsuccessful if it is negative. Descendants of *TWindowsObject*, including *TWindow* and *TDialog*, check *Status* before creating their associated elements. Use *Status* in the code of your descendant types to flag an initialization error.

Possible error values include *em_InvalidWindow*, *em_InvalidClient*, *em_InvalidChild*, and *em_InvalidMainWindow*.

TransferBuffer

```
TransferBuffer: Pointer;
```

Read/write

TransferBuffer points to a transfer record defined by an application that uses the transfer mechanism. Otherwise, it is **nil**.

Methods

Init

```
constructor Init(AParent: PWindowsObject);
```

Override: Often

Constructs and initializes the interface object. Constructors of descendant types must call *TWindowsObject.Init*. Calls *EnableAutoCreate* so that child windows will, by default, be created and displayed along with their parent windows. Also, adds the interface object to the child-window list of its parent-window object.

See also: *TWindowsObject.EnableAutoCreate*, *TWindowsObject.AddChild*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads an interface object from the stream *S* by reading the *Status*, other attributes, and the size of *ChildList* and then loading each child window.

Done

```
destructor Done; virtual;
```

Override: Often

Disposes of the interface object by first destroying the associated interface element, if any, and calling the *Done* destructor inherited from *TObject*. Disposes of all its child windows and removes itself from its parent's child-window list. Destructors of descendant types must include a call to *TWindowsObject.Done*.

AddChild

```
procedure AddChild(AChild: PWindowsObject);
```

Adds *AChild* to the object's child-window list.

At

```
function At(I: Integer): PWindowsObject;
```

Returns a pointer to the *I*th child in the object's child-window list. The child-window list is circular, so if *I* is greater than the number of child windows, *At* wraps around.

CanClose

```
function CanClose: Boolean; virtual;
```

Override: Sometimes

Calls the *CanClose* method for each child window and returns *False* if any child window returns *False*, indicating that it is not OK to close the interface element. If all child windows' *CanClose* methods return *True*, *CanClose* returns *True*.

ChildWithID

```
function ChildWithId(Id: Integer): PWindowsObject; virtual;
```

Override: Never

Returns a pointer to the window in the child window list with the passed ID. If no child window matches, *ChildWithID* returns **nil**. If *Id* is –1, *ChildWithID* returns the first noncontrol object in the child-window list.

CloseWindow

```
procedure CloseWindow;
```

Calls *CanClose* to see if the window is ready to close. If *CanClose* returns *True*, *CloseWindow* disposes of the window object and destroys the associated window element.

See also: *TWindowsObject.CanClose*

CMExit

```
procedure CMExit(var Msg: TMessage); virtual cm_First + cm_Exit;
```

Responds to a *cm_Exit* command message by causing the application to terminate, if a call to *CanClose* returns *True*. The *cm_Exit* message will normally only be sent to the application's main window.

Create

```
function Create: Boolean; virtual;
```

Override: Never

Creates the interface object's associated screen element. This is an abstract method which is overridden in descendant types.

CreateChildren

```
function CreateChildren: Boolean;
```

Calls *Create* for all child windows. *CreateChildren* is called by *SetupWindow*, so you don't normally need to call it directly. *CreateChildren* only needs to be called after *GetChildren*, in order to create visual elements for child window objects loaded from a stream.

See also: *TWindowsObject.GetChildren*

CreateMemoryDC

```
function CreateMemoryDC: HDC;
```

Creates a memory device context (DC) compatible with the window's display device. Memory DCs are required for bitmap manipulation.

DefChildProc

```
procedure DefChildProc(var Msg: TMessage); virtual;
```

Override: Sometimes

Performs the default processing for an incoming child-ID-based message by setting the *Result* field of *Msg* to zero, indicating that the message was not processed.

DefCommandProc

```
procedure DefCommandProc(var Msg: TMessage); virtual;
```

Override: Sometimes

Performs the default processing for an incoming command-based message by setting the *Result* field of *Msg* to zero, indicating that the message was not processed.

DefNotificationProc

```
procedure DefNotificationProc(var Msg: TMessage); virtual;
```

Override: Sometimes

Performs the default processing for an incoming notification message by setting the *Result* field of *Msg* to zero, indicating that the message was not processed.

DefWndProc

```
procedure DefWndProc(var Msg: TMessage); virtual;
```

Override: Never

This default window procedure does nothing and is usually overridden. It relies on the *Result* field of *Msg* to remain zero, indicating that the message was not processed. *TWindow* overrides *DefWndProc* to call Windows-supplied default responses to Windows messages.

See also: *TWindow.DefWndProc*

Destroy

```
procedure Destroy; virtual;
```

Override: Never

Forces the destruction of the interface object's associated element, removing it from the screen, by causing the window object to receive a *wm_Destroy* message. *Destroy* also calls *EnableAutoCreate* for any child window that has been created. This ensures that, if recreated, the interface object will look like it did when destroyed.

See also: *TWindowsObject.WMDestroy*, *TWindowsObject.EnableAutoCreate*

Disable

```
procedure Disable;
```

Disables the screen element associated with the interface object by calling *EnableWindow*. A disabled screen element usually appears as grayed and receives no keyboard or mouse input from Windows. All screen elements are enabled by default but can be disabled by calling *Disable*.

See also: *TWindowsObject.Enable*

DisableAutoCreate

```
procedure DisableAutoCreate;
```

Disables the feature that allows the interface object, as a child window, to be created and displayed along with its parent window. Call *DisableAutoCreate* for pop-up windows and controls if you wish to create and display them at a time later than their parent windows.

See also: *TWindowsObject.EnableAutoCreate.*

DisableTransfer

```
procedure DisableTransfer;
```

Disables, for the interface object, the transfer mechanism, which allows a control's state information to be transferred to and from a transfer buffer.

DispatchScroll

```
procedure DispatchScroll(var Msg: TMessage); virtual;
```

Override: Never

Called by *WMHScroll* and *WMVScroll* to dispatch window scrolling messages to the appropriate objects.

See also: *TWindowsObject.WMHScroll*, *TWindowsObject.WMVScroll*

Enable

```
procedure Enable;
```

Enables the screen element associated with the object by calling *EnableWindow*. The screen element (and therefore the object) only receives keyboard and mouse input when enabled. Since screen elements are enabled by default, you only need to call *Enable* to enable a window that has been disabled.

See also: *TWindowsObject.Disable*

EnableAutoCreate

```
procedure EnableAutoCreate;
```

Ensures that the interface object, as a child window, is created and displayed along with its parent window. This feature is enabled, by default, for windows and controls, but disabled for dialogs. Call *EnableAutoCreate* if you want to create and display a dialog box along with its parent window.

See also: *TWindowsObject.DisableAutoCreate*

EnableKBHandler

```
procedure EnableKBHandler;
```

Enables a feature of windows and modeless dialog boxes that allows them to provide a keyboard interface to child controls, much like that provided by modal dialog boxes. This allows the user to tab through controls, for example. By default, this feature is disabled.

EnableTransfer

```
procedure EnableTransfer;
```

Enables, for the interface object, the transfer mechanism, which allows a control's state information to be transferred to and from a transfer buffer.

FirstThat

```
function FirstThat(Test: Pointer): PWindowsObject;
```

Iterates over the child-window list and calls the Boolean function pointed to by *Test*, passing each child-window object in turn as an argument. *FirstThat* returns a pointer to the first child-window object that returns *True* from the Boolean function (or **nil** if none returns *True*) and then stops iterating. For example, you can write a method, *GetFirstChecked*, that uses *FirstThat* to retrieve the first child check box in a checked state:

```
function MyWindow.GetFirstChecked: PWindowsObject;

  function IsThisOneChecked(ABox: PWindowsObject); Boolean; far;
  begin
    IsThisOneChecked := ABox^.GetCheck <> 0;
  end;

begin
  GetFirstChecked := FirstThat(@IsThisOneChecked);
end;
```

Focus

```
procedure Focus;
```

Tells Windows to give the input focus to the screen element associated with the object.

ForEach

```
procedure ForEach(Action: Pointer);
```

Iterates over the child-window list and, for each child window, calls the procedure pointed to by *Action* and passes the child-window object as an argument. For example, you can write a method, *CheckAllBoxes*, that uses *ForEach* to check every check box in the child-window list:

```
procedure MyWindow.CheckAllBoxes;

  procedure CheckTheBox(ABox: PWindowsObject); far;
  begin
    PCheckBox(ABox)^.Check;
  end;

begin
  ForEach(@CheckTheBox);
end;
```

GetChildPtr

```
procedure GetChildPtr(var S: TStream; var P);
```

Loads a child-window pointer *P* from the stream *S*. *GetChildPtr* should only be used inside a *Load* constructor to read pointer values that were written by a call to *PutChildPtr* from a *Store* method.

See also: *TWindowsObject.GetSiblingPtr*, *TWindowsObject.PutSiblingPtr*, *TWindowsObject.PutChildPtr*

GetChildren

```
procedure GetChildren(var S: TStream);
```

Reads child windows from the given stream and puts them in the window's child-window list. *GetChildren* assumes that *ChildList* is initially empty; pointers to children added prior to calling *GetChildren* will be lost.

See also: *TWindowsObject.CreateChildren*, *TWindowsObject.PutChildren*

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Sometimes

Returns the default window class name, 'TurboWindow'.

GetClient

```
function GetClient: PMDIClient; virtual;
```

Override: Never

Returns **nil** for all nonMDI interface objects, which have no MDI client windows. *TMDIWindow* overrides this method to supply its MDI client window.

GetID

```
function GetId: Integer; virtual;
```

Override: Seldom

In general, returns the window ID. By default, *GetID* simply returns –1. *GetId* is redefined by *TControl* to return the object's control ID. All other interface objects have no control IDs.

See also: *TControl.GetId*

GetSiblingPtr

```
procedure GetSiblingPtr(var S: TStream; var P);
```

Loads a sibling-window pointer *P* from the stream *S*. A *sibling window* is a window with the same parent as this window—a *TCheckBox*'s *TGroupBox*, for example, is a sibling of the *TCheckBox* in a dialog. *GetSiblingPtr* should only be used inside a *Load* constructor to read pointer values that were written by a call to *PutSiblingPtr* from a *Store* method. The value loaded into *P* does not become valid until the window's parent completes its *Load* operation; therefore, dereferencing a sibling-window pointer within a *Load* constructor does not produce the correct result.

See also: *TWindowsObject.PutSiblingPtr, TWindowsObject.GetChildPtr, TWindowsObject.PutChildPtr*

GetWindowClass

```
procedure GetWindowClass(var AWndClass:TWndClass); virtual;
```

Override: Sometimes

Serves as a place holder for descendant types to define the window class record and return it in *AWndClass*. This *GetWindowClass* does nothing.

IndexOf

```
function IndexOf(P: PWindowsObject): Integer;
```

Returns the ordinal position of *P* in the object's child-window list. The first child window in the list is numbered 1. Returns zero if *P* is not in the child-window list.

IsFlagSet

```
function IsFlagSet(Mask: Byte); Boolean;
```

Returns the state of the bit flag in the *Flags* field specified by the value in *Mask*. *IsFlagSet* returns *True* if the bit flag is set, and *False* if it is clear.

See also: *TWindowsObject.SetFlags*

Next

```
function Next: PWindowsObject;
```

Returns a pointer to the next object in the parent window's child-window list.

See also: *TWindowsObject.Previous*

Previous

```
function Previous: PWindowsObject;
```

Returns a pointer to the previous object in the parent window's child-window list.

See also: *TWindowsObject.Next*

PutChildPtr

```
procedure PutChildPtr(var S: TStream; var P: PWindowsObject);
```

Store a child-window pointer *P* on the stream *S*. *PutChildPtr* should only be used inside a *Store* method to write pointer values that can later be read by a call to *GetChildPtr* from a *Load* constructor.

See also: *TWindowsObject.GetSiblingPtr*, *TWindowsObject.PutSiblingPtr*, *TWindowsObject.GetChildPtr*

PutChildren

```
procedure PutChildren(var S: TStream);
```

Iterates through the window's child-window list, writing each of the child windows to the given stream. *PutChildren* is called automatically by *TWindowsObject.Store*, but it can also be called directly in cases where you want to save the contents of a window without storing the window itself.

See also: *TWindowsObject.GetChildren*

PutSiblingPtr

```
procedure PutSiblingPtr(var S: TStream; P: PWindowsObject);
```

Stores a sibling pointer *P* on the stream *S*. A sibling window is a window with the same parent as this window. *PutSiblingWindow* should only be used inside a *Store* method to write pointer values that can later be read by a call to *GetSiblingPtr* from a *Load* constructor.

See also: *TWindowsObject.GetSiblingPtr*, *TWindowsObject.GetChildPtr*, *TWindowsObject.PutChildPtr*

Register

```
function Register: Boolean; virtual;
```

Override: Never

Registers the window class defined in the object's *GetWindowClass* method and named in its *GetClassName* method if it is not already registered. *Register* returns *True* if the class is registered successfully.

RemoveChild

```
procedure RemoveChild(AChild: PWindowsObject);
```

Removes *AChild* from the object's child-window list.

SetFlags

```
procedure SetFlags(Mask: Byte; OnOff: Boolean);
```

Turns a bit flag in the *Flags* field on or off, depending on the value of *OnOff*. If *OnOff* is *True*, the bit in *Mask* is set. Otherwise, the bit is cleared. *Mask* can be any of the *wb_* constants, or a combination of them.

See also: *TWindowsObject.IsFlagSet*

SetupWindow

```
procedure SetupWindow; virtual;
```

Override: Often

Initializes the newly created screen element, usually by creating child-window elements, if any. It creates only those child windows whose

auto-create feature is enabled. By default, this includes windows and controls, but not dialogs. If *Create* is unable to create a child window, it sets *Status* to *em_InvalidChild*. *SetupWindow* also calls *TransferData* to copy data into the new child windows.

Show

```
procedure Show(ShowCmd: Integer); virtual;
```

Override: Never

Show displays the interface element on the screen in a manner specified by the value passed in *ShowCmd*. The allowable values for *ShowCmd* include:

Table 21.28
Show method parameter values

Value	Description
sw_Hide	Hidden
sw_Show	In the window's current size and position
sw_ShowMaximized	Maximized and active
sw_ShowMinimized	Minimized and active
sw_ShowNormal	Restored and active

Store

```
procedure Store(var S: TStream);
```

Stores the interface object on the stream *S* by writing *Status*, the other attributes, and the size of *ChildList*. Each child window is then stored.

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
```

Override: Sometimes

Returns zero. *Transfer* is redefined by *TControl* descendants to transfer their state data to and from the transfer buffer. The return value from *Transfer* is the number of bytes of data transferred.

TransferData

```
procedure TransferData(Direction: Word); virtual;
```

Override: Sometimes

If the transfer mechanism is enabled by setting *TransferBuffer* to a transfer record, transfers data from to or from the buffer and the interface object's participating child windows. *TransferData* calls the *Transfer* method of each participating child window and passes a pointer to the transfer buffer as well as the direction specified in *Direction*. *Direction* can be *tf_SetData* or *tf_GetData*.

See also: *TWindowsObject.EnableTransfer*, *TWindowsObject.DisableTransfer*, *TWindowsObject.SetupWindow*

WMActivate

```
procedure WMActivate(var Msg: TMessage); virtual wm_First + wm_Activate;
```

Override: Sometimes

In the case that the interface object is participating in the keyboard handling mechanism, responds to the interface object becoming the active window by calling the application object's *SetKeyboardHandler* method.

See also: *TApplication.SetKeyboardHandler*

WMClose

```
procedure WMClose(var Msg: TMessage); virtual wm_First + wm_Close;
```

Override: Sometimes

Responds to a request to close the window by calling this object's *CanClose* method, or the application object's *CanClose* method in the case that this object is the application's main window. If *CanClose* returns *True*, this interface element is destroyed by calling *Destroy*.

See also: *TWindowsObject.Destroy*

WMCommand

```
procedure WMCommand(var Msg: TMessage); virtual wm_First + wm_Command;
```

Override: Seldom

Provides the mechanism for handling command-based and child-ID-based messages and calling the appropriate response methods.

WMDestroy

```
procedure WMDestroy(var Msg: TMessage); virtual wm_First + wm_Destroy;
```

Override: Seldom

If this object is the application's main window, *WMDestroy* responds to this interface element's destruction by informing Windows that the application is ending, resulting in a *wm_Quit* message. If the object is not the application's main window, the default window behavior is invoked.

WMHScroll

```
procedure WMHScroll(var Msg: TMessage); virtual wm_First + wm_HScroll;
```

Override: Seldom

Intercepts horizontal window scroll bar messages and calls *DispatchScroll*.

See also: *TWindowsObject.DispatchScroll*

WMNCDestroy

```
procedure WMNCDestroy(var Msg: TMessage); virtual wm_First + wm_NCDestroy;
```

Override: Never

Responds to the last message an interface element receives before destruction by setting *HWindow* to zero.

WMQueryEndSession

```
procedure WMQueryEndSession(var Msg: TMessage); virtual wm_First +
  wm_QueryEndSession;
```

If the window is the application's main window, it responds to the *wm_QueryEndSession* message by calling *Application^.CanClose*, and if that returns *True*, sets *Msg.Result* to 1; otherwise, it sets *Msg.Result* to 0.

WMVScroll

```
procedure WMVScroll(var Msg: TMessage); virtual wm_First + wm_VScroll;
```

Override: Seldom

Intercepts vertical window scroll bar messages and calls *DispatchScroll*.

See also: *TWindowsObject.DispatchScroll*

TWordArray type

Declaration

```
TWordArray = array[0..16383] of Word;
```

Function

A word array type for general use.

voXXXX constants — Validate

Function Constants beginning with *vo* represent the bits in the bitmapped *Options* word in validator objects.

Values The validator *Options* bits are defined as follows:

Figure 21.1 Validator option flags

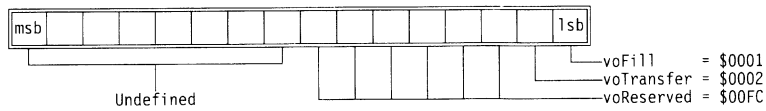


Table 21.29 Validator option flags

Constant	Value	Meaning
voFill	$0001	Used by picture validators to indicate whether to fill in literal characters as the user types
voTransfer	$0002	The validator handles data transfer for the input line. Currently only used by range validators.
voReserved	$00FC	The bits in this mask are reserved by Borland.

vsXXXX constants — Validate

Function Input line objects use *vsOK* to check that their associated validator objects were constructed properly. When called with a command parameter of *cmValid*, an input line object's *Valid* method checks its validator's *Status* field. If *Status* is *vsOK*, the input line's *Valid* returns *True*, indicating that the validator object is ready to use.

The only value defined for *Status* other than *vsOK* is *vsSyntax*, used by *TPXPictureValidator* to indicate that it could not interpret the picture string passed to it. If you create your own validator objects, you can define error codes and pass them in the *Status* field.

Values The *Validate* unit defines two constants used by validator objects to report their status:

Table 21.30 Validator status constants

Constant	Value	Meaning
vsOK	0	Validator constructed properly
vsSyntax	1	Error in the syntax of a picture validator's picture

See also *TValidator.Status*

wb_XXXX constants — OWindows

Function The *Flags* field in *TWindowsObject* is a bitmapped field. The bits can be accessed with constants beginning with *wb_*.

Values The following values are defined:

Table 21.31 TWindowsObject bitmapped field constants

Constant	Value	Meaning if set
wb_KeyboardHandler	$01	Window handles key events like a dialog
wb_FromResource	$02	Dialog was loaded from a resource
wb_AutoCreate	$04	Window is created when parent window is created
wb_MDIChild	$08	Window is an MDI child window
wb_Transfer	$10	Window participates in the *Transfer* mechanism. By default, this bit is set by *InitResource*, cleared by *Init*.

See also *TWindowsObject.Flags*

wm_XXXX constants — OWindows

Function ObjectWindows defines several constants related to the standard Windows messages, defining ranges of messages reserved for Windows.

Values The following constants are defined:

Table 21.32 Window message constants

Constant	Value	Meaning
wm_First	$0000	Beginning of Windows messages
wm_Count	$8000	Number of Windows messages

WordRec type — Objects

Declaration

```
WordRec = record
  Lo, Hi: Byte;
end;
```

Function A utility record allowing access to the *Lo* and *Hi* bytes of a word.

See also *LongRec*

ws_XXXX Window styles — WinTypes

You can use these constants to set the creation attributes of an interface object by combining them in the interface object's *Attr.Style* field before the object creates its screen element. You can also use them to specify window styles when creating windows, dialog boxes and controls with *CreateWindow* or *CreateWindowEx*.

Table 21.33 Window styles

Constant	Meaning
ws_Border	The window has a border. Not valid with *ws_DlgFrame*.
ws_Caption	The window has a title bar and a border. *ws_Caption* and *ws_DlgFrame* cannot be used together. *ws_Caption* implies the inclusion of *ws_Border*.
ws_Child	The window is a child window. *ws_Child* and *ws_Popup* cannot be used together.
ws_ChildWindow	Same as *ws_Child*
ws_ClipChildren	The window does not include the area covered by its child windows when drawing.
ws_ClipSiblings	The window clips all sibling windows when drawing. This means that drawable regions in each client region of child windows of the same parent will not overlap. Only valid with *ws_Child*.
ws_Disabled	The window is initially disabled.
ws_DlgFrame	The window has a double border and no title. Not valid with *ws_Border*.
ws_Group	The window is a control which is the first of a group of controls the user can access with arrow keys. Each subsequent control defined without *ws_Group* belongs to the group started by the last control with *ws_Group*.
ws_HScroll	The window has a horizontal scroll bar.
ws_Iconic	Same as *ws_Minimize*
ws_Maximize	The window appears full screen (maximized).
ws_MaximizeBox	The window has a maximize box.
ws_Minimize	The window is initially minimized (iconic). Use only with *ws_Overlapped*.
ws_MinimizeBox	The window has a minimize box.
ws_Overlapped	The window is an overlapped window. An overlapped window has a caption and a border.
ws_OverlappedWindow	The same as *ws_Overlapped*, *ws_Caption*, *ws_SysMenu*, *ws_ThickFrame*, *ws_MinimizeBox*, and *ws_MaximizeBox* combined.
ws_Popup	The window is a pop-up window. Not valid with *ws_Child*.
ws_PopupWindow	The same as *ws_Border*, *ws_Popup*, and *ws_SysMenu* combined. The Control-menu box appears only if the *ws_Caption* style is used also.
ws_SizeBox	Same as *ws_ThickFrame*
ws_SysMenu	The window has a Control-menu box in its title bar. It applies only to windows with title bars.

Table 21.33: Window styles (continued)

ws_TabStop	The window is a dialog box control in a list of controls which the user can cycle through using *Tab*.
ws_ThickFrame	The window has a thick frame the user can drag to resize the window.
ws_Tiled	Same as *ws_Overlapped*
ws_TiledWindow	Same as *ws_OverlappedWindow*
ws_Visible	The window is initially invisible.
ws_VScroll	The window has a vertical scroll bar.

See also *TWindow.Attr*

I N D E X

B

D

E

F

G

H

M

N

O

P

R

S

T

U

V

W

X

Y

7.0

OBJECTWINDOWS™

BORLAND

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-8400. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom ▪ Part #11MN-BPL04-70 ▪ BOR 4686