Turbo Pascal® for Windows

Windows Programming Guide

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

R1

10 9 8 7 6 5 4 3

C O N T E N T S

T A B L E S

F I G U R E S

Windows without pain

Turbo Pascal for Windows provides an exciting new way to develop applications for Microsoft Windows. Until now, most Windows programming has used the C programming language and a lot of development tools that run under DOS. As a result, the process of developing Windows programs tended to be long, complicated, and confusing.

Enter, Turbo Pascal for Windows

But that was then, and this is now. Turbo Pascal for Windows introduces several important new innovations to take the pain out of developing Windows applications.

- Extensions of the Pascal language
- A Windows-hosted environment
- Windows-hosted, interactive resource editing tools
- The ObjectWindows library to streamline Windows programming
- Turbo Debugger for Windows

Let's look at each of these features.

The Turbo Pascal language

Up until now, if you've developed applications for Windows, you have probably been using the C language. But sometimes you just don't want to write in C, or you may have Pascal code you wish

to adapt to Windows. Turbo Pascal now has all the facilities you need to develop rich Windows applications.

In addition to all the features you've come to expect from Turbo Pascal, like fast compilation, reuseable code units, and object-oriented programming, Turbo Pascal for Windows also provides support for using and creating dynamic-link libraries (DLLs), direct inclusion of Windows resource files, several new data types, and revised versions of the standard DOS Turbo Pascal units for Windows.

Programming for Windows, in Windows

The first thing you'll probably notice about writing Turbo Pascal programs for Windows is that the integrated development environment (IDE) runs in Windows as a Windows program. This means you can write, compile, and test your Turbo Pascal applications while making full use of the multitasking capabilities of Windows.

Resource editing tools

Turbo Pascal for Windows includes the Microsoft resource compiler and a full set of resource editors. The resource editors enable you to edit resources such as menus, dialogs, bitmaps, icons, and cursors visually, without programming. The resource compiler lets you compile resource scripts written by other Windows programmers.

The ObjectWindows library

Windows introduces many new wrinkles you may never have had to think about before, like dealing with text and graphics in resizeable windows, interacting with other programs in a multitasking environment, and manipulating the nearly 600 functions in the Windows API. Perhaps the most frustrating part of all 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, you cannot write directly to the screen. Nor can you 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.

Turbo Debugger for Windows

Turbo Debugger for Windows is a powerful source-level debugger designed to work with Turbo Pascal for Windows. Turbo Debugger for Windows provides full support for evaluation of Turbo Pascal and assembler expressions, with special facilities for debugging Windows applications and object-oriented programs.

Full details for use of the debugger can be found in the *Turbo Debugger for Windows User's Guide*.

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 Turbo Pascal and Windows. The elements of programming in Turbo Pascal can be found in the *User's Guide* and the *Programmer's Guide*, and details on using Windows comes with the Windows software.

In addition, you need to be pretty 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 Chapter 4, "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.

What's in this book?

Because Turbo Pascal for Windows is a new approach to Windows programming, and because ObjectWindows utilizes some techniques you may not be familiar with, this manual includes a lot of explanatory material. Complete reference material for both ObjectWindows and the Windows Application Programming Interface can be found in the *Windows Reference Guide*.

This manual is divided into four parts:

- Part 1, *Learning ObjectWindows*, introduces you to the principles of writing Windows applications in Turbo Pascal, 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 graphics, use of resources, memory management, sharing code and data, and guidelines for Windows programming.
- Part 4, *Collections and streams*, describes how to use the collection and stream mechanisms with Turbo Pascal objects, with a special emphasis on uses in ObjectWindows.

PART 1

Learning ObjectWindows

C H A P T E R

1

Inherit the window

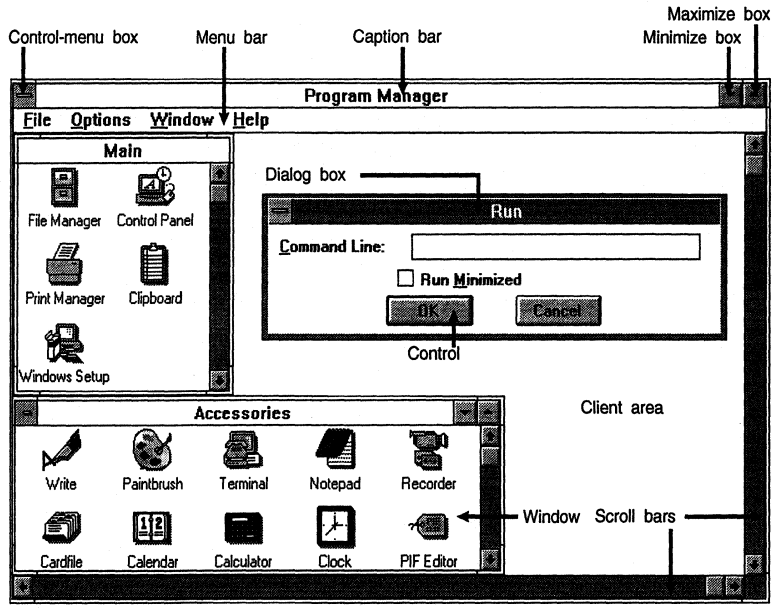
This chapter is an overview of programming for Microsoft Windows using Turbo Pascal, with an emphasis on object-oriented programming. The examples presented here use ObjectWindows supplied with Turbo Pascal. You will learn what goes into a Windows application and what comes out. You will learn the required behavior of a Windows application, and how object-oriented programming with ObjectWindows automates many tasks and simplifies others.

To get the most out of this chapter, you need to understand object-oriented programming concepts. If object-oriented programming is new to you, study Chapter 4, "Object-oriented programming," in the *User's Guide*. You should also know how to use Windows.

What is a Windows application?

Figure 1.1 shows the major components of a Windows application. This documentation depends on your knowledge of these components.

Figure 1.1
The onscreen components of a Windows application



A Windows application is a special type of PC program that

- Must be in a special executable (.EXE) file format.
- Runs only with Windows.
- Generally, runs in a rectangular window on the screen.
- Follows user interface guidelines to appear and perform in a standard way.
- Can run simultaneously with other Windows and non-Windows applications, including other instances of itself.
- Can communicate and share data with other Windows applications.

Benefits of Windows

Windows offers many benefits to both users and developers. Benefits to users include

- Standard and predictable operation. If you know how to use one Windows application, you know how to use them all.
- No need to set up devices and drivers for each application. Windows provides drivers to support peripherals.
- Inter-application cooperation and communications.

- Multitasking: the ability to run many applications at once.
- Access to more memory. Windows supports protected mode.

Benefits to developers include

- Device-independent graphics, so graphical applications run on all standard display adapters.
- Immediate support for a wide range of printers, monitors and mice.
- A rich library of graphics routines.
- More memory for large programs.
- Support for menus, icons, bitmaps and more.

Requirements

The other side to the array of benefits Windows offers to the user and developer is the more stringent list of hardware requirements. Windows generally requires better graphics, more memory, and faster processors for equivalent performance compared to a DOS application. If you have an 80286 machine or better and at least 2 MB of memory, Windows will run fine.

Programming facilities

Windows provides a great many programming facilities for the application developer.

An event-driven architecture

Windows is based on an event-driven architecture. This means that all user input is handled as *events*. Whether the event is clicking a mouse button or pressing a keyboard key, an event occurs and Windows generates a *message*. For example, if the user presses the left mouse button, Windows generates a message called *wm_LButtonDown*. If the user presses a key, Windows generates a *wm_KeyDown* message. Windows treats all menu and control commands, whether selected by mouse or keyboard, as *wm_Command* messages. You'll learn more about messaging in Chapter 7, "Object Windows overview." This event-driven architecture fits in nicely with Turbo Pascal's object-oriented approach.

Device-independent graphics

Windows unifies the process of writing to the screen and printer into a single module, called the graphics device interface (GDI), which provides a common interface to every Windows program. What's more, Windows supplies device drivers for most standard graphics adapters and printers. The resulting system allows you to write one application that runs, unmodified, on a great majority of the world's currently available hardware.

Device-independent graphics offers some benefits that might not be immediately apparent. For one, Windows applications are generally easy to install, since they don't have to reconfigure your system with their particular device drivers. Another is that Windows applications often run better on a local area network because each user has his or her own local configuration.

But device-independent graphics comes at a cost. The developer's cost is adhering to the somewhat strict requirements of GDI. GDI limits the programmer's options in designing applications.

Multitasking

Interprocess communication is covered in Chapter 16, "Dynamic Data Exchange."

Windows allows users to run many applications concurrently, eliminating the need for terminate-and-stay-resident (TSR) programs. Not only does Windows allow multitasking, it supports it with a number of facilities for interprocess communication, such as the Clipboard and dynamic data exchange (DDE).

Windows manages multiple applications by limiting each application's use of the full screen to one or more rectangles called *windows*. These windows can be moved, resized, and temporarily hidden as icons, allowing the user to switch between tasks quickly. From the programmer's side, this means that a program should not write text and graphics directly to screen locations. Instead, it should draw only into its window's *client area*, the area inside the window frame. Likewise, an application has to share the computer's available memory with other applications. A well-behaved Windows application correctly follows Windows' rules of screen and memory management.

Memory management

In a typical Windows session, multiple applications are started and closed many times, so it is not feasible to simply load each application into memory at the end of the previous one: Windows would soon run out of memory. Instead, Windows can *move* most of an application's memory, either to another part of memory or onto disk, to accommodate other applications or Windows itself.

Windows memory management is described in detail in Chapter 14, "Memory management."

A Windows application, therefore, must accommodate Windows' dynamic memory management by avoiding direct access of memory locations. For example, a traditional pointer to a memory location could soon become invalid when Windows reallocates memory, because the pointer may point to memory that is now being used for something else.

In place of pointers, Windows applications use *handles*, which are essentially pointers to pointers. Handles are numbers that serve as indexes into a table of pointers managed by Windows. Thus, Windows applications refer to a window or display context (an area for drawing on the screen) by its handle. There are also handles to application instances, strings, drawing tools, and resources such as menus and icons.

In normal usage, you won't have to deal with memory handles yourself. You can allocate and deallocate heap space using the usual *New*, *Dispose*, *GetMem*, and *FreeMem* routines, and Turbo Pascal deals with Windows to make sure it knows where those pointers actually point.

One of the primary advantages of Windows memory management is the ability to share compiled code among applications. For example, if the user runs two instances of the same application, the two applications use the same compiled code in memory. Likewise, an application can dynamically load a library module that can be shared among different applications. This is known as a dynamic-link library, or DLL.

Resources

Resources are descriptions of a Windows application's user interface devices: its menus, dialog boxes, cursors, icons, bitmaps, strings, and accelerator keys. Windows provides a facility for maintaining these descriptions outside of an application's source code. The application's resources are united with its compiled

executable file before the application is run. In order to limit memory usage, the application calls its resources into memory only when they are needed.

Separating resource specification from the source code has an added benefit: The look and feel of an application can be altered without affecting the program's source code. In fact, you need not even *have* the source code to modify an application's resources. This makes it easy to customize or translate existing Windows applications.

Turbo Pascal includes a full set of editors for creating and customizing resources.

Dynamic linking

Windows allows applications, including Turbo Pascal programs, to load and free library modules at run time. These modules must be in a special executable (EXE) format called a *dynamic-link library* (DLL). Often these libraries perform a specific and complex task such as file format conversions. If this is the case, an application can use DLLs as filters for file importing and exporting. What's more, DLLs can be shared among a group of applications, promoting reuse and memory conservation.

You can write dynamic-link libraries in Turbo Pascal.

Clipboard

The Windows Clipboard lets users transfer information such as text, graphics, and data between applications, between different parts of an application, or to temporary storage for later use. For example, a word processing application would use the Clipboard for its cut, copy, and paste operations.

Dynamic data exchange

Dynamic data exchange (DDE) is another information transfer protocol. Where the Clipboard is under the user's control, DDE works behind the scenes under the program's control. An application transfers information to another application by sending DDE messages.

Multiple document interface

Multiple document interface (MDI) is a set of user interface conventions for creating windows that contain child windows inside them. The Turbo Pascal integrated development environment (IDE) is an example of MDI. The user can open several edit windows on the Turbo Pascal desktop. Each edit window is a child window.

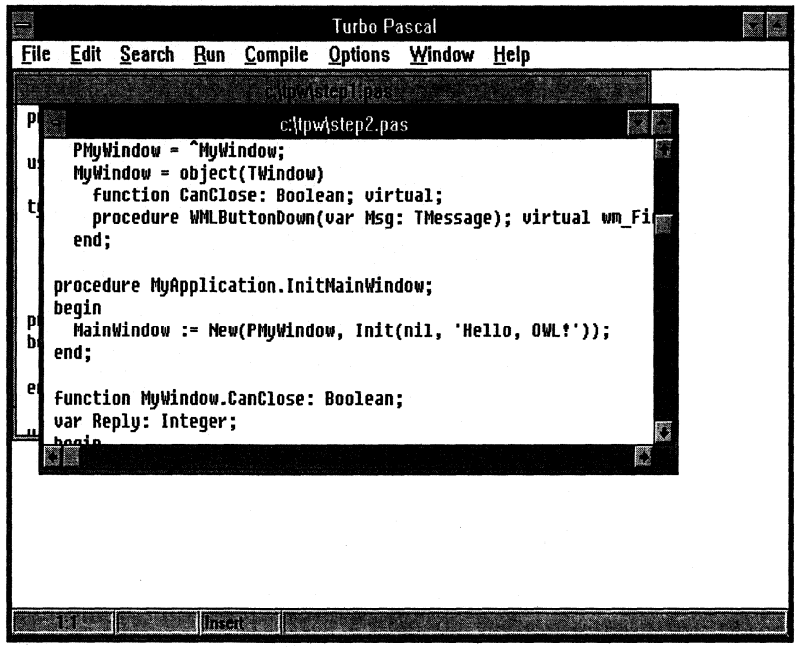


Figure 1.2
The Turbo Pascal IDE is an MDI application

Windows data types

Because of Windows' data management scheme and its association with the C programming language, programming for Windows with Turbo Pascal is greatly facilitated by some specialized data types. For example, a handle to a window is stored as an *HWnd* type. Turbo Pascal and ObjectWindows define new types to accomodate types such as *HWnd*. All these new types and data structures are documented in the *Windows Reference Guide*.

Object-oriented windowing

As you can see, programming for a windowing environment demands an awareness of many events, formats, handles, and other applications, so developing a Windows program can be a daunting task. Fortunately, object-oriented programming simplifies the task of programming for a windowing environment, allowing the application developer to focus on the application's function, rather than its form. By using objects to represent complex structures such as windows, a Turbo Pascal program can encapsulate its operations and data storage. This is the goal of ObjectWindows.

Object-oriented programming (OOP) provides a framework within which the programmer uses objects to represent the fairly complex user-interface elements of a Windows program. This means, most obviously, that a window is an object. The ObjectWindows window and application object types manage the message-processing behavior required from a Windows program, greatly simplifying the programmer's interaction with the user. In fact, ObjectWindows objects represent more than just windows: they represent dialog boxes and controls, such as list boxes and buttons.

A better interface to Windows

ObjectWindows uses object-oriented extensions of Turbo Pascal to encapsulate the Windows application programming interface (API), insulating you from the details of Windows programming. As a result, you can use Turbo Pascal to write Windows programs with much less time and effort than with non-object-oriented approaches. Specifically, ObjectWindows provides three helpful features: encapsulation of window information, abstraction of Windows API functions, and automatic message response.

Interface objects

While ObjectWindows defines objects for windows, dialog boxes, and controls, it supplies only the object's behavior, attributes, and data storage. The physical implementation, the item's visual appearance on the screen, is managed by Windows itself. Thus, ObjectWindows objects, which we'll call *interface objects*, form a partnership with the corresponding visual elements, which we'll call *interface elements*. Successful management of the object/

element partnership is the key to successful Windows programming with ObjectWindows.

The connection in the object/element relationship is the window handle. When you construct an interface object, one of the things it does is tell Windows to create an interface element. Windows returns a handle identifying that element, which the object stores in a field called *HWindow*. Many Windows functions require the window handle as a parameter, so storing it in a field keeps it readily accessible to the window object. Similarly, interface object fields can be used to store drawing tools or status information for that particular window.

Abstracting Windows functions

Windows applications affect their appearance and behavior by calling Windows functions, the set of almost 600 functions that makes up the Windows application program interface (API). Each function takes a variety of parameters of many different types, which can become quite a headache. Although you can call any Windows function directly from Turbo Pascal, ObjectWindows simplifies the task by offering object methods that abstract the function calls.

As noted earlier, many of the parameters for Windows functions are already stored in the fields of interface objects. Thus, methods can use this data to supply Windows functions with parameters. In addition, ObjectWindows groups related function calls into single methods that perform higher-level tasks. The result is a streamlined, easier-to-use API to enhance the existing Windows API.

While this approach greatly reduces your dependence on the hundreds of Windows API functions, it does not restrict you from calling the API directly. ObjectWindows offers the best of both worlds: high-level, object-oriented development plus maximum control over the graphical environment.

Automating message response

In addition to telling the Windows environment to do things, most applications need to be able to respond to the hundreds of Windows messages that result from user actions (such as clicking the mouse), other applications, or other sources. Processing and responding to messages correctly is critical to the proper functioning of your program. After all, your application must respond in *some* way to a menu selection, and responding to any particular message is no big deal. But writing an application that

knows how to respond to close to 200 different Windows messages can be as daunting as calling the right Windows functions.

Objects, with their predefined behavior (methods), are perfectly suited to the task of responding to incoming stimuli (Windows messages). ObjectWindows takes Windows messages and turns them into Turbo Pascal method calls. Therefore, using ObjectWindows, you simply define a method to respond to each message your application needs to handle. For example, when the user presses the left mouse button down, Windows generates a *wm_LButtonDown* message. If you want a window or control in your program to respond to such mouse clicks, you simply define a *WMLButtonDown* method keyed to the *wm_LButtonDown* message. Then, whenever Windows sends that message, your object will automatically call the method you've defined.

Such methods are called *message response methods*. Without object-oriented programming and ObjectWindows, you would be required to write a lengthy **case** statement for each window and control, recognizing that a message has arrived, then sorting out what kind of message it is, then deciding what to do with it. ObjectWindows takes care of all that for you.

The structure of a Windows program

With so many software elements such as DOS, Windows, and applications interacting at once, it helps you to know about how parts of your Windows applications interact with the world around them. This section explores the structure of Windows and typical Windows applications written in Turbo Pascal with ObjectWindows.

The structure of Windows

At run time, the functionality of Windows and its API resides in three external library modules called by the currently running applications. Here are the Windows modules:

- KERNEL.EXE — Responsible for memory and resource management, scheduling, and interaction with DOS.
- GDI.EXE — Responsible for displaying graphics on the screen and printer.

- USER.EXE — Responsible for window management, user input, and communications.

These modules are a part of the retail version of Windows, so they are already loaded on your users' computers, assuming they have Windows. You will supply a program that makes use of these library modules, but does not actually include them.

Interacting with Windows and DOS

Because of the limited scope of the DOS operating system, it is easy to overlook the contribution DOS makes to the successful operation of your DOS application programs. Nonetheless, a DOS program runs because of the interaction between your application code and the facilities of the operating system. The same is true of a Windows program. Because Windows offers so many more operating system functions, it is harder to overlook the interplay between Windows and an application. For example, to draw graphics on the screen, your program must call a Windows GDI function. To respond to a user clicking the mouse, your program must define a message response method. Your program must continually interact with the operating system (DOS plus Windows).

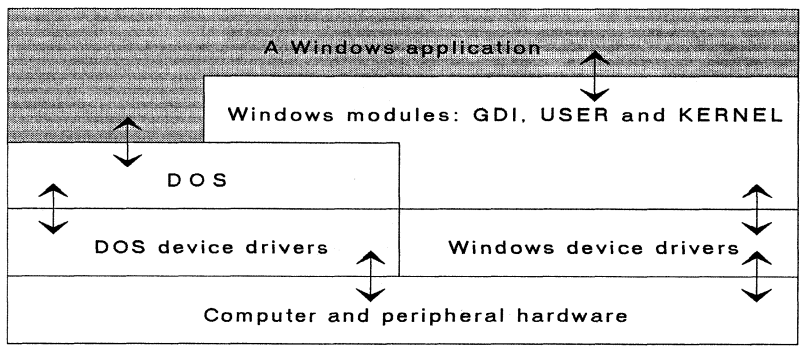


Figure 1.3
How Windows applications interact with Windows and DOS. The shaded part is what you write.

"Hello, Windows"

The traditional way to introduce a new language or environment is to present a "Hello, World" program written in the language or for the environment. This program usually consists of only enough code to display the string "Hello, World" on the screen.

Of course, in Windows there's a lot more to do than that. You need to put up a window, write in it, and then make the window understand how to interact with the world around it, at least enough for you to close the window and make it go away. If you do this from scratch, it takes quite a bit of code just to get those basic tasks done. For example, the program GENERIC.PAS, included on your distribution disks, performs just these minimal tasks, and it's well over 100 lines long!

That's because Windows has a list of requirements an application must meet before it can run in Windows. Even the simplest program requires a substantial amount of code. Fortunately, programs written with ObjectWindows automatically meet most of those requirements, including creating and displaying the main window and storing a handle to the application. Therefore, "Hello World" is simplified to just 16 lines:

```
program HelloApp;

uses WObjects;

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

procedure THelloWorld.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil, 'Hello, Turbo Pascal world'));
end;

var HelloWorld: THelloWorld;

begin
  HelloWorld.Init('HelloWorld');
  HelloWorld.Run;
  HelloWorld.Done;
end.
```

Application startup responsibilities

An ObjectWindows program's first act upon starting is to take four values from Windows and store them in global variables. This happens automatically and is something you would have to handle if you wrote applications that didn't use ObjectWindows. Here are the global variables.

- *HInstance* — stores a handle to the application instance.
- *HPrevInst* — stores a handle to the last instance of the same application. It's zero if this is the first instance.

- *CmdShow* — stores an integer representing the initial main window display mode. It's used for calls to the *Show* method.
- *CmdLine* — stores a string representing the application startup command line, including options and file name. For example, 'CALC.EXE /M' or 'WORDPROC.EXE LETTER1.DOC'.

As an ObjectWindows application, *HelloApp* must construct and initialize the main window object. It can initialize only the first instance of *HelloApp* with the *InitApplication* method, or it can initialize each instance of *HelloApp* with the *InitInstance* method.

HelloApp starts the message loop by calling *Run*. Finally it ends itself by disposing of the application object using the *Done* method.

Main window responsibilities

The main window of an application is the window that first appears when an application is started. It is responsible for presenting to the user a list of available commands (a menu). During the course of the application session, the main window manages the application's interface, and in many cases, serves as the program's only working area, spawning dialog boxes where appropriate. Other, more complex applications might have many windows that serve as work areas. Finally, when the user closes the main window, it is responsible for initiating the process to close the application.

The application development cycle

Since there are certain requirements of any Windows application (initializing the main window, for one), it is usually easiest to begin your application by using an existing Windows application and customize it from there. ObjectWindows supplies many sample programs. Choose the one most like your application.

Using the integrated development environment within Windows, you can save a great deal of development time. Because of Windows' multi-tasking ability, you can run the IDE, resource editors, the debugger, and your application — all at the same time. Not only do the tool supplied with Turbo Pascal for Windows make each task easier, but they also cut down on the number of tasks in developing a Windows application. Essentially, the process boils down to just these steps:

1. Create the program code.
2. Create resources for dialogs, menus, etc.

3. Compile the program (resources can be linked in automatically).
4. Debug the program interactively.

Turbo Pascal for Windows provides the easiest environment for developing Windows applications.

C H A P T E R

2

Stepping through Windows

Now that you've been briefly introduced to the ObjectWindows library, you're ready to start building some simple ObjectWindows programs. 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 even a simple help system. On the way, you will be introduced to the major principles of Windows application design, such as message processing, managing parent and child window relationships, and automatic graphics redrawing.

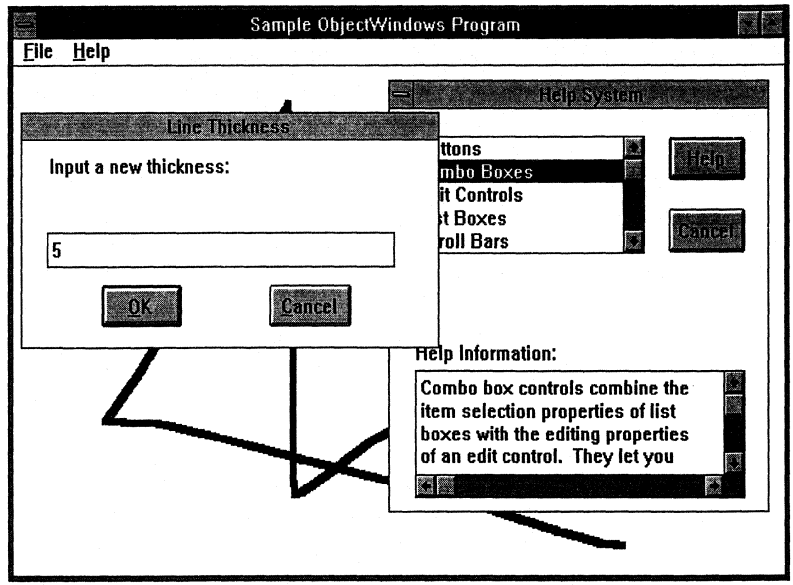
This walk-through consists of ten steps.

- Step 1: Creating an application
- Step 2: Defining a main window object type
- Step 3: Drawing text in a window
- Step 4: Drawing lines in a window
- Step 5: Changing line thickness
- Step 6: Repainting your windows
- Step 7: Adding a menu
- Step 8: Adding popup windows and dialog boxes
- Step 9: Storing the drawing in a file
- Step 10: Adding controls in a window and using units

The source code for the application is provided at various stages on the distribution disks. The files are named STEP1.PAS, STEP2.PAS and so on, corresponding to the steps in the tutorial.

Figure 2.1 shows the application you will have created at the end of this cookbook.

Figure 2.1
A complete ObjectWindows application



Step 1: A simple Windows application

You'll start your application development by writing a bare-bones ObjectWindows application, called *MyProgram*. This program, found in STEP1.PAS, can serve as the starting point for all of the programs you write in ObjectWindows. *MyProgram* will instantiate and create the application's main window.

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 a 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 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 *MyProgram*, this type is called *TMyApplication*. Here is the main block of *MyProgram*:

```
var
  MyApp: TMyApplication;
begin
  MyApp.Init('MyProgram');
  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 'MyProgram'. *Init* also creates and shows the application's main window. *Run* sets off 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 inherited method, *InitMainWindow*. *TApplication.InitMainWindow* is an abstract method that is called automatically by ObjectWindows to set up your program's main window. Every ObjectWindows application *must* construct its own main window to do anything meaningful.

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

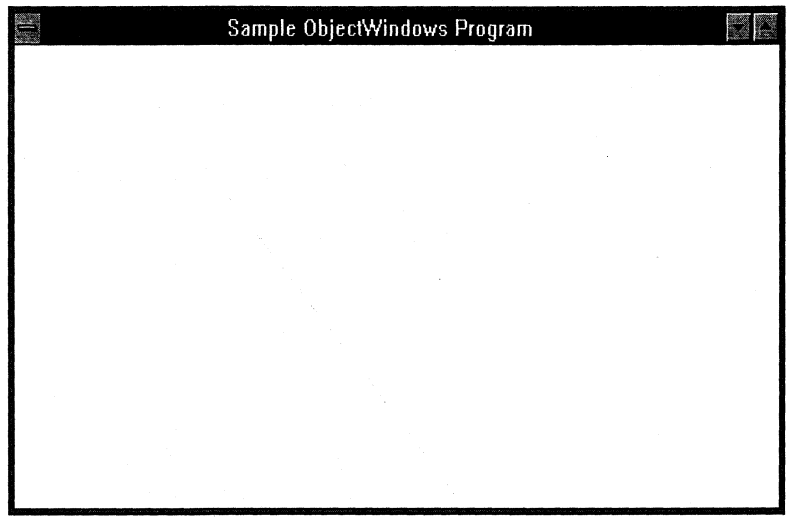
```
TMyApplication = object(TApplication)
  procedure InitMainWindow; virtual;
end;
```

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 hierarchically. This ownership relationship is called an *instance linkage*.

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

You will usually modify the above 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, you will replace it with a more interesting window type. Figure 2.2 shows the appearance of *MyProgram*.

Figure 2.2
Your first ObjectWindows program, *MyProgram*



To write an ObjectWindows program, you will have to use at least the *WObjects* unit. More complicated applications will need to use additional units. For example, any program that makes calls directly to the Windows API will have to include the units *WinProcs* and *WinTypes*.

At this point, *MyProgram* does nothing but display a blank window that can be moved, resized, maximized, minimized and closed. Here is a full listing of *MyProgram*, up to this point:

This is STEP1.PAS.

```
program MyProgram;

uses WObjects;

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

procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil,
    'Sample ObjectWindows Program'));
```

```
end;

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

Step 2: The main window object type

The program you created in Step 1 consists of two objects: an application object and a window object. The application object, *MyApp*, is an instance of type *TMyApplication*, a type you derived from *TApplication*. The window object, held in the *MainWindow* field of the application object, 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 will bring the main window object to life by defining a more specialized window type, derived from *TWindow*.

What is a window object?

In Step 1 you saw that an application object encapsulates the standard behaviors of a Windows application, including construction of the main window. Type *TApplication* provides the fundamental behaviors of every application you will create.

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

Handles

All window objects have at least three fields: *HWindow*, *Parent*, and *ChildList*. As noted in Chapter 1, *HWindow* holds the handle to the window. A handle is an ID number that associates an interface object, such as a window, dialog box, or control object, with its corresponding interface element.

Thus *HWindow* holds an integer that identifies the appropriate interface 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 will not have to directly manipulate the window handle. It is needed, however, when calling Windows functions directly. For example, to bring up a message box, as you'll do later in this part, you will call the *MessageBox* function. *MessageBox* requires that you supply a parameter identifying the message box's parent window. You will supply the main window, whose handle is stored in the *HWindow* object field:

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

Parents and children

Most windows do not exist independently of others: They 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.

The *Parent* field holds 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 8.

The third window object field is *ChildList*, which holds a linked list of the window's *child windows*, if any. You will add child windows to your program in Step 8.

Creating a new window type

Now that you have some idea of what a window object holds, you can define a new window type, descending from type *TWindow*, to serve as a main window for *MyProgram*. First, update the type definitions to specify the new type, *TMyWindow*. Also, be sure to

define a pointer to the type, *PMyWindow*, which will be useful when you instantiate *TMyWindow*.

```
type
  PMyWindow = ^TMyWindow;
  TMyWindow = object(TWindow)
  end;
```

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

```
procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PMyWindow, Init(nil,
    'Sample ObjectWindows Program'));
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 amost 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 10, "Window objects."

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

Responding to messages

The quickest way to make a window object useful is to teach it how to respond to Windows messages. For example, when the user clicks the left mouse button in the main window of *MyProgram*, the corresponding window object receives a *wm_LButtonDown* message from Windows. 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 6 of this tutorial.

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

To intercept and respond to Windows messages, you must define a method for each type of incoming message you care to respond to. These are called *message response methods*. To mark a method definition header as a response method, you'll add an extension, which is the identifier of the message to which it is a response. For example, the method defined next will respond to all *wm_LButtonDown* messages:

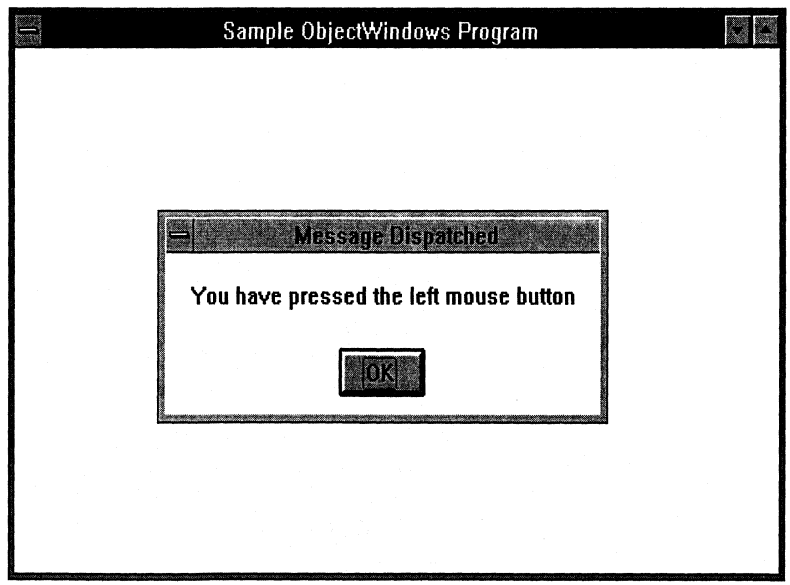
```
TMyWindow = object(TWindow)
  procedure WMLButtonDown(var Msg: TMessage); virtual wm_First +
    wm_LButtonDown;
end;
```

Msg is a *TMessage* record that holds information such as the coordinates of the point that was clicked on. You'll examine the *Msg* argument further in Step 3.

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

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

Figure 2.3
MyProgram responding to a user event



The full source code for this step is listed at the end of this chapter.

Terminating an application

The program created here closes 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 may not be for some others.

For example, have you ever quit an application without saving your work? A good application always asks if the user wants to save work, if it has not yet been saved, before quitting. You can easily add this behavior to your ObjectWindows applications. Take *MyProgram* and add the ability to double-check the user's request to quit.

When the user tries to close your ObjectWindows application, the virtual *CanClose* method of your application type is invoked. *CanClose* is a Boolean function that indicates whether it is okay (*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 okay to close.

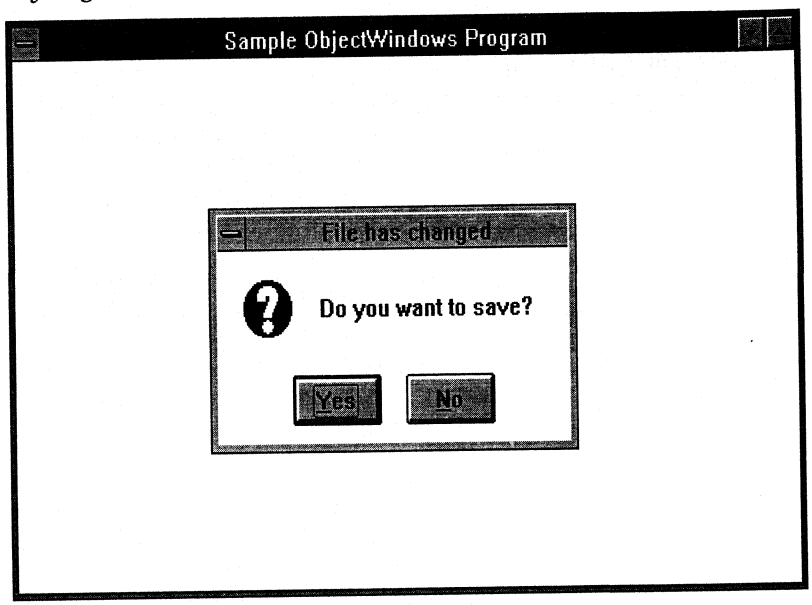
Your main window type, *TMyWindow*, inherits a *CanClose* method from *TWindowsObject* that calls the *CanClose* methods of each of its child windows, if any. If there are no child windows (as in this case), *CanClose* simply returns *True*. To modify an application's closing behavior, you'll redefine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

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

Now, when users try to close *MyProgram*, they are presented with a message box that asks, "Do you want to save?" Clicking on the Yes button returns *False* and prevents the main window and application from closing. Clicking on the No button returns *True* and the application terminates. In Part 4, you will give this

message box some meaning. Figure 2.4 shows the modified *MyProgram*.

Figure 2.4
MyProgram with refined closing behavior



Here is the full source code to *MyProgram* thus far:

This is STEP2.PAS.

```
program MyProgram;

uses WinTypes, WinProcs, WObjects;

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

type
  PMyWindow = ^TMyWindow;
  TMyWindow = object(TWindow)
    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;

function TMyWindow.CanClose: Boolean;
var
  Reply: Integer;
begin
  CanClose := True;
```

```
    Reply := MessageBox(HWindow, 'Do you want to save?',
      'File has changed', mb_YesNo or mb_IconQuestion);
    if Reply = id_Yes then CanClose := False;
  end;

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

  procedure TMyWindow.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(PMyWindow, Init(nil, 'Sample ObjectWindows
  Program'));
  end;

  var
    MyApp: TMyApplication;

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

C H A P T E R

3

Filling in the window

Menus, dialog boxes, and popup windows manage the program's user interface. However, the program doesn't do anything interesting yet. In this part, you'll take *MyProgram*, 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 *MyProgram*'s main window. Then you'll transform *MyProgram* into a full graphics application that lets you draw lines into the main window. After that, you will refine the drawing program to redraw its graphics, change the thickness of the lines, and finally, save its graphics into a file for reloading at a later time.

What in the world is a display context?

Device contexts are described in detail in Chapter 17, "All about GDI."

You can 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. Windows manages a display context in its own memory space, but the application keeps track of it by storing a handle to a display context. Like a handle to a window, a handle to a display context is a number that identifies the correct Windows display context.

Since a display context is for drawing in a window, you will create a new object field for your main window object, called *DragDC*, to hold the handle to a display context. *DragDC* will be

of type *HDC*, a special Windows-style type equivalent to the standard *Word* type in Turbo Pascal.

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);
```

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);
```

After drawing text or graphics, you *must* release the display context. It is very important that you release any display context you have obtained as soon as you are done drawing.

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

If you do not release all obtained display contexts, you will soon run out of them (the entire Windows environment has a limit of five), and your application will fail, usually causing your computer to hang. If your application fails the third or fourth time you draw something, unreleased display contexts are the first thing to check.

The display context serves some important drawing functions. First, it ensures that you do not draw your text and graphics outside the surface of a window. Second, it manages the selection and deletion of drawing tools: pens, brushes, and fonts. An example of selecting a new pen is provided in the graphics drawing example later in this part. But first, you'll start with drawing text.

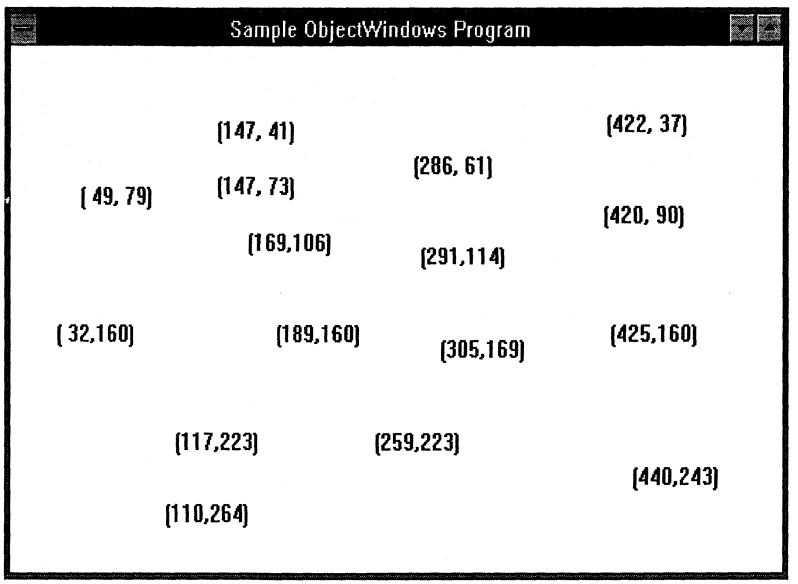
Step 3: Drawing text in a window

To draw text in the main window of *MyProgram*, follow the display context drawing model introduced in the preceding section; first display context, and finally release the display context. To make things interesting, draw the text in response to the left mouse button clicks you originally intercepted in Step 2. Instead of bringing up a message box, this time you'll respond by

drawing text that shows the coordinates of the point where you clicked on 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'll also draw in right on the clicked point. This example also has the benefit of familiarizing you with the Windows coordinate system. See Figure 3.1.

Figure 3.1
MyProgram drawing text where the user clicks



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

Message records

In Step 2, you saw that the *Msg* argument of a message response method carries valuable information, 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 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 will use these fields: *wParam*, *lParamLo*, and *lParamHi*.

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 will use the Windows function *WVSPrintF* to format the string. *WVSPrintF* is explained in Chapter 2 of the *Windows Reference Guide*.

Note that Windows expects strings to be null-terminated (ending with a zero byte). See Chapter 13 in the Programmer's Guide for details on using these strings.

Once you have obtained the final string, you can draw it at the point that was clicked by passing back to Windows, in a *TextOut* function call, the string, *S*, the coordinates, *Msg.lParamLo* and *Msg.lParamHi*, and the string length, *StrLen(S)*. Also, be sure to obtain the display context before drawing and release it afterwards.

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

The calls to *Str* and *StrPCopy* do the type conversions. *Str* converts the coordinates, of type *Word*, to string types. *StrPCopy* (a procedure found in the *Strings* unit) converts a Pascal-style string to a null-terminated string.

Clearing the screen

One more function you can add to the text drawing application is clearing the screen. 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.

You'll clear the window in response to a right mouse button click. To implement this, 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 repain itself, it just clears its client area:

```
procedure TMyWindow.WMRButtonDown(var Msg: TMessage);
```

```
begin
  InvalidateRect(HWindow, nil, True);
end;
```

You can see the current source code in the file STEP3.PAS.

Step 4: Drawing lines in the window

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.

You will take the following steps:

1. Respond to left mouse button clicks and drags by connecting the dots along the way, resulting in drawn lines.
2. Respond to the right mouse button clicks by bringing up an input dialog, allowing the user to change the line thickness.
3. Make the window automatically redraw its contents by storing the points and redrawing them in response to a *paint* message.

To do these steps, first you'll study the Windows dragging model, then you'll implement a simple graphics drawing program.

The dragging model

It will be helpful to first review the Windows mouse event model. You have already seen that a left mouse button click results in a *wm_LButtonDown* message and a *WMLButtonDown* method call. Earlier in this tutorial, you responded to left mouse button clicks by bringing up message boxes and 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 screen. But these responses only cover the initial clicks of a mouse button. Many interactive Windows programs require you 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.

You will do this by responding to a few more messages. *wm_MouseMove* is received when the user drags the mouse to a new point in a window and *wm_LButtonUp* is received when the user releases the left mouse button. Typically, a window will receive 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 will respond to *wm_LButtonDown* by initiating the drawing process (obtaining a display context, among other things). It will respond to *wm_MouseMove* by drawing or moving graphics, and it will respond to *wm_LButtonUp* by terminating the drawing process (releasing the display context).

The following table summarizes the most common mouse event messages.

Table 3.1
Common mouse event messages

Message	Event
wm_LButtonDown	The user clicks down on the left mouse button.
wm_RButtonDown	The user clicks down on the right mouse button.
wm_MouseMove	The user drags the mouse.
wm_LButtonUp	The user clicks up on the left mouse button.
wm_RButtonUp	The user clicks up on the right mouse button.
wm_LButtonDblClk	The user double-clicks the left mouse button.
wm_RButtonDblClk	The user double-clicks the right mouse button.

Responding to drag messages

The following table summarizes how you'll respond to drag messages to create your line drawing program:

Table 3.2
Messages used in Step 4

Message	Response
wm_LButtonDown	Clear the screen. Then obtain a display context and store it in *DragDC*. Situate the drawing pen at the point clicked.
wm_MouseMove	Draw a line from the previous point to the current point.
wm_LButtonUp	Release *DragDC*.

As stated previously, *wm_LButtonDown* is always followed by *wm_LButtonUp*, with or without *wm_MouseMove* messages in between. After all, what goes down must come up! Therefore, every time you obtain a display context, you later release it. In

this case, you'll obtain a single display context for all the drawing that takes place between the time the user clicks down on the left mouse button and the time he releases it.

Pairing the obtaining with a release is critical to the proper functioning of your graphics programs. However, you can also add one more safety measure. Define a new Boolean object field for *TMyWindow*, the main window type, called *ButtonDown*. *WMLButtonDown* will set it to *True* and *WMLButtonUp* will set it 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 TMyWindow.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 TMyWindow.WMMouseMove(var Msg: TMessage);
begin
  if ButtonDown then
    LineTo(DragDC, Msg.lParamLo, Msg.lParamHi);
end;

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

MoveTo and *LineTo* are graphics functions in the Windows API which 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 are not drawing directly on the window, but on its display context.

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, it will still send *wm_MouseMove* messages to the main window, rather than to an adjacent window.

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

This brings the code up to STEP4.PAS.

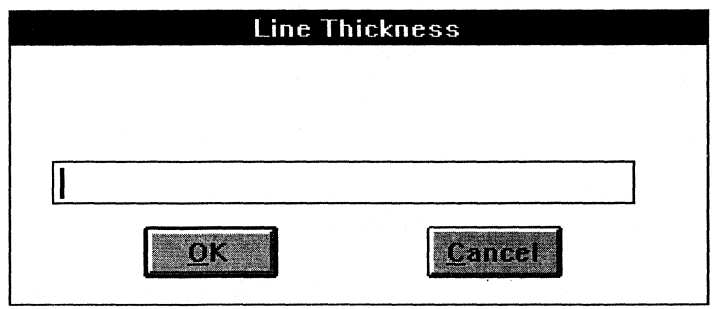
```
procedure WMLButtonUp(var Msg: TMessage); virtual wm_First +
  wm_LButtonUp;
procedure WMMouseMove(var Msg: TMessage); virtual wm_First +
  wm_MouseMove;
```

Step 5: Changing the line thickness

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

You'll also use an input dialog (of type *TInputDialog*) to provide a mechanism for the user to change the line thickness. Figure 3.2 shows the input dialog in use.

Figure 3.2
Changing line thickness



Tracking line thickness

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.

Earlier, you saw the drawing tools used to produce graphics and text in a window: pens, brushes, and fonts. These drawing tools are window elements, whose descriptions are stored in Windows memory, not unlike interface elements. Also like windows, drawing tools are accessed by applications by means of handles. Since drawing tools are not represented by ObjectWindows objects, your programs are responsible for creating them and for deleting them from Windows memory when you are done with them.

Think of a drawing tool as a painter's paintbrush and a display context as his canvas. Once he has created his drawing tools (paintbrush), he obtains a display context (canvas), and selects the proper drawing tools. Similarly, a Windows program must select drawing tools *into* a display context. 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 will select a different, thicker pen for drawing in the window.

The first step is to add support for input dialogs to *MyProgram*. Add the unit *StdDlgs* to the **uses** statement. In order to use the Windows-compatible string manipulation functions, also use *Strings*. The start of your program file should now look like this:

```
program MyProgram;

uses Strings, WinTypes, WinProcs, WObjects, StdDlgs;
...
```

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

```
type
  PMyWindow = ^TMyWindow;
  TMyWindow = object(TWindow)
    DragDC: HDC;
    ButtonDown: 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;
```

To initialize these new data 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 TMyWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  ButtonDown := False;
  PenSize := 1;
  ThePen := CreatePen(ps_Solid, PenSize, 0);
end;

destructor TMyWindow.Done;
begin
  DeleteObject(ThePen);
  TWindow.Done;
end;
```

Now alter the *WMLButtonDown* method to select the current pen (*ThePen*) into the newly obtained display context. If *ThePen* equals zero, there has been no special pen created, so use the default (thin) pen. Like *MoveTo* and *MessageBox*, *SelectObject* is a Windows API function.

```
procedure TMyWindow.WMLButtonDown(var Msg: TMessage);
begin
  InvalidateRect(HWindow, nil, True);
  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:

```
procedure TMyWindow.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.

Running the input dialog

Clicking the right mouse button is a convenient way to bring up the option to change the line thickness. Let's redefine the *WMRButtonDown* method to bring up an input dialog, one of ObjectWindows' stock dialogs. An input dialog serves as a simple dialog box that takes one line of text input. To use it, you need not modify *TInputDialog* or any of its methods.

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

Once the input dialog object has been constructed with *Init*, you can run it as a modal dialog by calling the *Execute* method. *Execute* is similar to *Create* for a window object in that it creates the interface object's corresponding element. The processing for *Execute*, however, ends only after the user has closed the dialog by clicking OK or Cancel. If the user clicks OK, *InputTxt* 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 TMyWindow.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;
```

As one final step, be sure to delete the final pen from Windows memory before terminating the application. To do this, override the *Done* destructor to delete the pen.

```
destructor TMyWindow.Done;
begin
  DeleteObject(ThePen);
  TWindow.Done;
end;
```

Step 6: Automatically 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. Where did they go?

A better question is, "Where were they in the first place?" You never stored the text or lines in any type of variable. Once the graphics data goes to Windows through calls to Windows functions, you can never get it back to redraw it when needed. In order to have a window re-display its graphics, you've got to store the graphics in some other type of structure; an object is well suited to the task. Objects can store simple or complex graphics, and can be easily maintained as object fields of the main window.

The painting model

When the user of your application resizes or uncovers your window, it requires updating, or *painting*. 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.

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 will be the same as when they were drawn the first time by the user, much like replaying an audio recording of a concert; Is it live or is it ObjectWindows? But first, you need to store the graphics as objects, so you can paint them in a *Paint* method.

Storing graphics as objects

Let's store the drawn line as a collection of points to be stored in a *TMyWindow* object field called *Points*. For this collection, you will use the type *TCollection*, found in the *WObjects* unit. At all times, *Points* will contain the current drawing in the form of a collection of points. Whenever the window needs painting, it will use the data stored in *Points* to replay the drawing of the line.

The next question to answer is, "What is a line?" You saw in step 4 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 an object type to represent points. Define *DPoint* to hold x- and y-coordinates as object fields.

Because *TCollection* objects know how to grow dynamically as you add more elements, they are well suited to holding an unknown quantity of lines and points. In fact, one line might hold thousands of points.

Here is the object definition for *DPoint*:

```
type
  PDPoint = ^TDPoint;
  TDPoint = object(TObject)
    X, Y: Integer;
    constructor Init(AX, AY: Integer);
  end;

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

While *DPoint* does not define any new behavior, it needs to be an object in order to be stored on a stream later on. Be sure to construct *Points* in *TMyWindow.Init* and dispose of it in *TMyWindow.Done*. To do this, you'll have to override the *Init* and *Done* inherited from *TWindow*.

```
constructor TMyWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  ButtonDown := False;
  ThePen := CreatePen(ps_Solid, 1, 0);
  PenSize := 1;
  Points := New(PCollection, Init(50, 50));
end;

destructor TMyWindow.Done;
begin
  TWindow.Done;
  DeleteObject(ThePen);
  Dispose(Points, Done);
end;
```

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

In order to store the lines as objects, you must alter *WMLButtonDown* and *WMMouseMove*.

```
procedure TMyWindow.WMLButtonDown(var Msg: TMessage);
var
  APoint: PDPoint;
begin
```

```
    Points^.DeleteAll;
    InvalidateRect(HWindow, nil, True);
    if not(ButtonDown) then
    begin
      ButtonDown := True;
      SetCapture(HWindow);
      DragDC := GetDC(HWindow);
      if ThePen <> 0 then SelectObject(DragDC, ThePen);
      MoveTo(DragDC, Msg.lParamLo, Msg.lParamHi);
      Points^.Insert(New(PDPoint, Init(Msg.LParamLo, Msg.LParamHi)));
    end;
  end;

  procedure TMyWindow.WMMouseMove(var Msg: TMessage);
  var APoint: PDPoint;
  begin
    if ButtonDown then
    begin
      LineTo(DragDC, Msg.lParamLo, Msg.lParamHi);
      Points^.Insert(New(PDPoint, Init(Msg.LParamLo, Msg.LParamHi)));
    end;
  end;
```

WMLButtonUp requires no modification.

Redrawing stored graphics

Now that *TMyWindow* 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 *TMyWindow* that repeats the actions of *WMLButtonDown, WMMouseMove,* and *WMLButtonUp*. By iterating over the point collection, *Paint* sees the points in the order that the user originally entered them. Here is the *Paint* method:

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var  First: Boolean;

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

begin
  SelectObject(PaintDC, ThePen);
  First := True;
  Points^.ForEach(@DrawLine);
end;
```

C H A P T E R

4

Adding a menu

Most Windows applications have a menu on their main window to provide a variety of selections for the user, such as File | Save, File | Open, and Help. In this section, you'll add a standard menu to *MyProgram*.

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:

1. Design the menu as a menu resource.
2. Load the menu resource into the main window object.
3. Define responses to menu selections.
4. Specify the resource file from the program.

Menu resources

Somewhere in an application with a menu, there must be a menu specification that contains the text of the menu selections and the structure of the top-level items and their sub-items. This is true of any ObjectWindows application, but this specification is not part of the program's Pascal source code. It's part of a separate specification called a *resource*. Windows stores resources in a compact and efficient way. You will use the Resource Toolkit to inter-

actively design menus and other resources, such as dialog boxes, icons, and bitmaps.

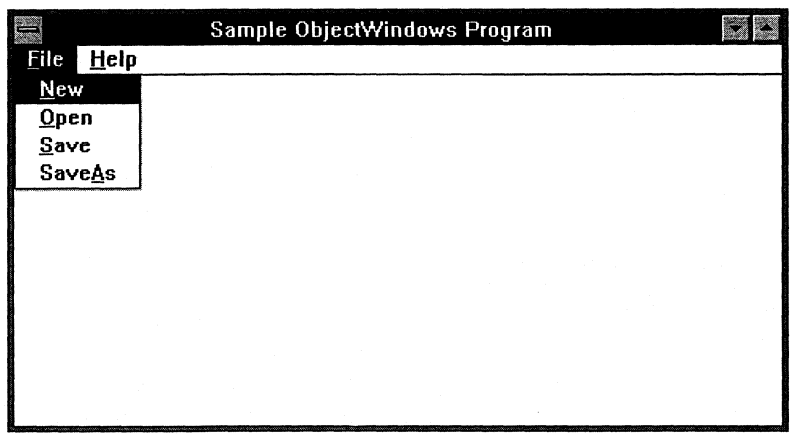
An application accesses its attached resources by specifying the resource ID. This ID is an integer, such as 100, or integer constant, such as *MyMenu*. An application distinguishes one menu selection from another by the menu ID associated with each menu item.

To continue with *MyProgram*, use a resource editor or the resource compiler to create a menu resource and save it as a .RES file, COOKBOOK.RES. See the file COOKBOOK.RC for the resource file in source format. You can also use the COOKBOOK.RES file provided on the distribution disks. Once the .RES file is created, include it using the **$R** compiler directive like this:

```
{$R COOKBOOK.RES}
```

Figure 4.1 shows the appearance of this menu (resource ID 100), including Help (menu ID 901) and File selections, the latter with the sub-items New, Open, Save, and SaveAs (menu IDs 101, 102, 103, and 104, respectively). Top-level menus that have sub-items, such as File, do not have menu IDs, and selecting them causes no action besides displaying their sub-items. If a top-level menu item has no sub-items, as in Help, it has a menu ID and can cause some action.

Figure 4.1
MyProgram with a menu resource



Because the menu is designed outside of the ObjectWindows program and because menu handling is fairly simple, you don't need to make a menu into an object. Instead, make it an attribute of the main window, much like the main window's caption. In the next step, you will attach the menu to the main window.

Step 7: A menu for the main window

As stated previously, an application's menu is not a separate object, owned by the main window. It's not an object at all, but merely an attribute of the main window. In fact it is stored in the *Menu* field of *Attr*, a window object field that stores a record of a window's *creation attributes*. To set the menu attribute, among others, you will redefine the *Init* constructor for your window type, *TMyWindow*.

The menu resource stored in COOKBOOK.RES has a resource ID of 100. Obtain the resource by calling the *LoadMenu* Windows function:

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

PChar(100) casts the number 100 into a string 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. To obtain access to *PChar*, the **$X+** compiler directive must be on (which is the default setting).

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 *TMyWindow.Init* should look. Notice that the first thing you did was to call *TWindow.Init* to perform the initialization required of all window objects:

```
constructor TMyWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(100));
  ButtonDown := False;
  PenSize := 1;
  ThePen := CreatePen(ps_Solid, PenSize, 0);
  Points := New(PCollection, Init(50, 50));
end;
```

Now, when the main window is displayed, it has the operational menu shown in Figure 4.1. In order to make the menu selections do something, however, you must follow the remaining steps to intercept and respond to the menu message.

Intercepting the menu message

When the user selects a menu, the window to which the menu is attached receives a Windows *command message*. To process one of these messages, define a method for the *TMyWindow* object type using a special extension:

```
procedure FileNew(var Msg: TMessage); virtual cm_First + 101;
```

Message ranges and offsets are explained more thoroughly in Chapter 7.

where *cm_First* is a ObjectWindows-defined constant defining the beginning of the range of constants for commands, and 101 is the desired menu ID. Do not confuse this dynamic method index, based on *cm_First*, with the one for responding to incoming Windows messages, which is based on the offset *wm_First*. This is a special case only for menus and accelerators.

To make the code more readable, substitute the menu IDs with constants you'll define at the beginning of the program:

```
const
  cm_New = 101;
  cm_Open = 102;
  cm_Save = 103;
  cm_SaveAs = 104;
  cm_Help = 901;
```

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

```
procedure FileNew(var Msg: TMessage); virtual cm_First + cm_New;
procedure FileOpen(var Msg: TMessage); virtual cm_First + cm_Open;
procedure FileSave(var Msg: TMessage); virtual cm_First + cm_Save;
procedure FileSaveAs(var Msg: TMessage); virtual cm_First +
  cm_SaveAs;
procedure Help(var Msg: TMessage); virtual cm_First + cm_Help;
```

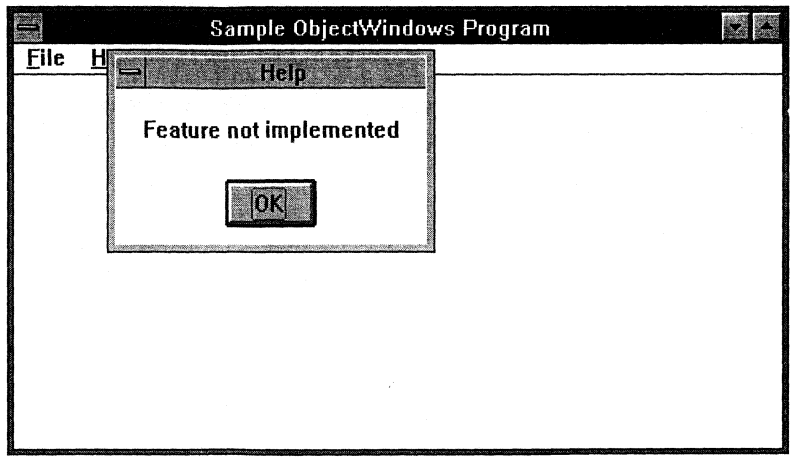
Responding to the menu message

For each menu selection, you now have a method that will be invoked. For example, for a Help selection, your *Help* method will be invoked. For now, you'll merely display a message box:

```
procedure TMyWindow.Help(var Msg: TMessage);
begin
  MessageBox(HWindow, 'Feature not implemented.', 'Help', mb_OK);
end;
```

Figure 4.2 shows *MyProgram*'s response to the Help selection.

Figure 4.2
MyProgram with help system



At this point, you can respond to the File | New menu selection in an interesting way, by clearing the screen. Add the following *FileNew* method:

```
procedure TMyWindow.FileNew(var Msg: TMessage);
begin
  Points^.FreeAll;
  InvalidateRect(HWindow, nil, True);
end;
```

This method deletes all of the stored points and forces a repainting of the screen. Since there are no points to redraw, the screen becomes blank.

For *FileOpen*, *FileSave*, and *FileSaveAs*, write dummy methods similar to *Help*. Later you will rewrite these methods to perform meaningful actions.

The complete source code to *MyProgram* up to this point is in the file STEP7.PAS.

Attaching resources to the executable file

After you have compiled your ObjectWindows program, but before you run it, you must attach the menu resource. Using the **$R** compiler directive, this is done automatically at the end of the

compile and link. The resource file specified in the directive is appended to the compiled executable file.

Resources can also be added to or removed from existing executable files, and existing resources can be modified. For information on how to do this, refer to the *Whitewater Resource Toolkit*.

C H A P T E R

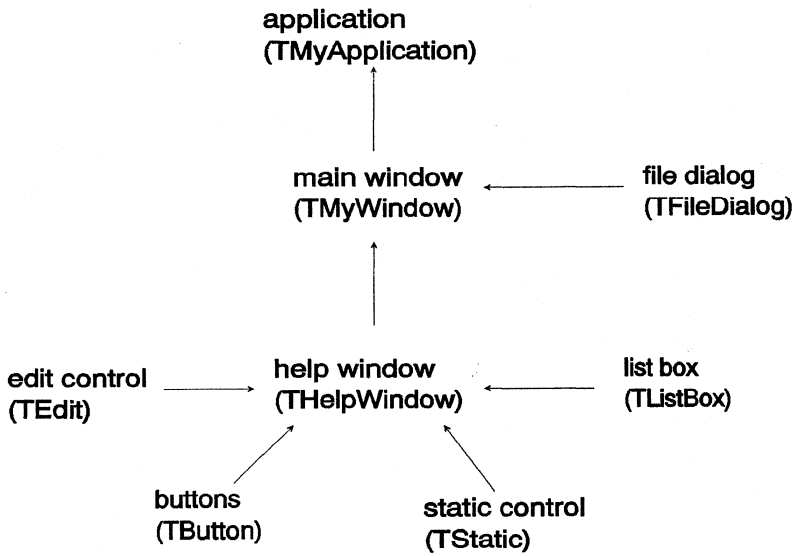
5

Holding a dialog

Up to this point, you have created only two types of windows, main windows (the *TMyWindow* object) and message boxes (for announcing "Feature not implemented" and "Are you sure you want to quit?"). A full-featured Windows program could have ten or more window objects (windows, controls, and dialogs) associated with it. With the exception of the main window, each window has one parent window upon which it is dependent. These dependent windows are called child windows. For example, when the user closes a parent window, its child windows also close.

These parental relationships result in a network of related parent and child windows for each application, with the main window as the ultimate parent. In this network, most windows serve as a child to one window and as a parent to others, and no window is an orphan. The group of related child and parent windows for *MyProgram*, which is now only partly implemented, is shown in 5.1.

Figure 5.1
A group of related parent and child windows



There are two types of child windows. One type is an independent child window. This type controls its own appearance and location. Message boxes and pop-up windows are independent child windows. The other type is a dependent child window. This type appears to be stuck to the surface of its parent and moves around with the parent. Controls, such as the buttons on message boxes, and some other types of child windows are dependent child windows. In this step, you'll create two types of independent child window objects: pop-up windows and dialog boxes.

Do not confuse the terms parent and child with ancestor and descendant types. Parent and child refer to ownership. For example, the main window owns the help window, a window you'll begin to build in this part. This ownership concept in Windows is almost identical to the ownership, or instance-linkage, concept of object-oriented programming. In fact, throughout this tutorial, child window objects will generally be stored in data fields of their parent window objects. Although this is not required, it is a good design technique and results in clear and easily maintained programs.

Step 8: Adding a pop-up window

Let's add to *MyProgram* a pop-up window that appears as a result of selecting the Help menu. This window can serve as the basis of a help system for *MyProgram*. For now, you'll make the pop-up window an object instance of *TWindow*. In Step 10, you'll create a new type for it and give it some useful behavior. For now, just get it on the screen.

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

In this step, you'll alter the response to the Help menu selection to create and display a help window.

Creating and showing the pop-up window

In Step 7, you responded to the Help menu selection by putting up a message box. Here, you'll substitute that message box with a blank pop-up window object, *HelpWnd*. While the immediate results will be less interesting than the message box, you'll add a lot more behavior to the help window in Step 10.

For now, just redefine *Help* to create and display the help window:

```
procedure TMyWindow.Help(var Msg: TMessage);
var HelpWnd: PWindow;
begin
  HelpWnd := New(PWindow, Init(@Self, 'Help System'));
  with HelpWnd^.Attr do
  begin
    Style := Style or ws_PopupWindow or ws_Caption;
    X := 100;
    Y := 100;
    W := 300;
    H := 300;
  end;
  Application^.MakeWindow(HelpWnd);
end;
```

Figure 5.2 shows the new help window which appears when the user clicks on Help.

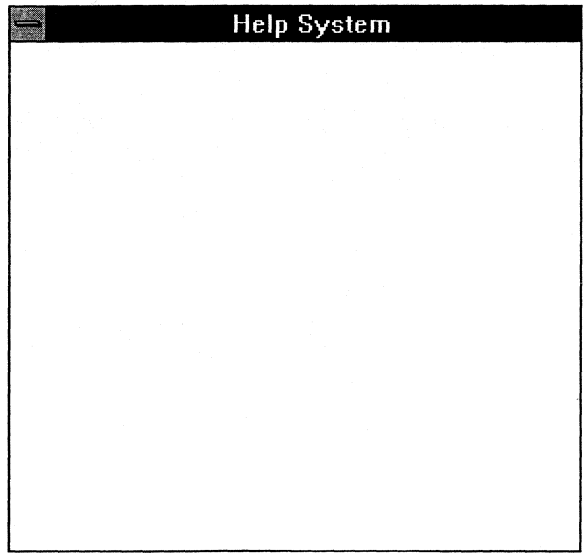


Figure 5.2
MyProgram's new help window

The MakeWindow function

The *TApplication* object defines a very important method called *MakeWindow*. *MakeWindow* performs all the actions necessary to associate an interface element with a window object safely. By "safely," we mean that *MakeWindow* checks for error conditions *before* they can cause serious problems like hanging your system. There are two important steps taken by *MakeWindow*.

ValidWindow is explained in more detail in the "Writing safe programs" section of Chapter 19.

The first is a test of the validity of the interface object, through a call to *ValidWindow*, which checks to make sure your application didn't run out of memory or otherwise fail to construct itself completely.

If the interface object was constructed successfully, *MakeWindow* attempts its second step: creating an interface element with the object's *Create* method. *Create* uses the information in the object's fields to tell Windows what kind of interface element you want. If the element cannot be created, an error message box will pop up.

If either the validity check or the interface element creation fails, *MakeWindow* returns **nil**. Otherwise, it returns a pointer to the interface object passed as its parameter.

Calling *MakeWindow* is the safest way to create interface elements. Calling *Create* directly without checking memory and trapping errors is dangerous, and not advisable.

Adding a dialog box

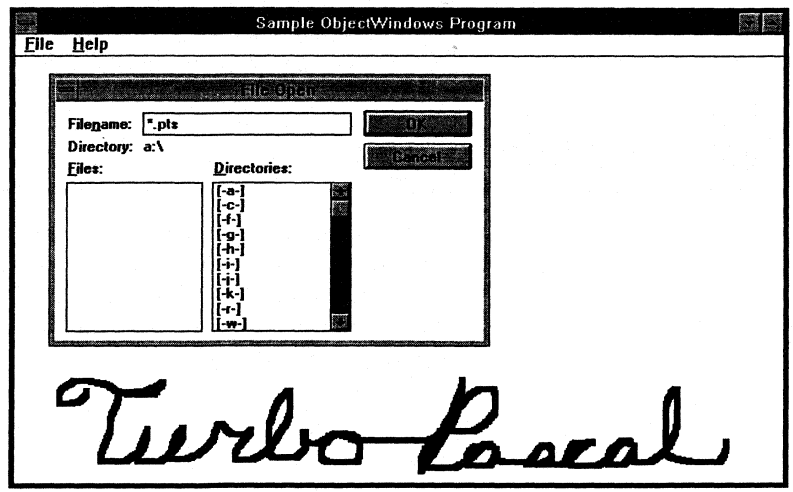
A dialog box is like a pop-up window, but it usually stays on the screen for a short period of time and performs one particular input-related task, such as choosing a printer or setting up a document page. Here, you'll add to *MyProgram* a dialog box for opening and saving files.

Like a pop-up window, a dialog box is an independent child window. Conceptually, adding a dialog box as a child window is exactly like adding a pop-up window, as you did in the previous section. After all, a dialog box is a lot like a window, but it does have some major differences:

- Dialog object types descend from type *TDialog* rather than *TWindow*. Both *TDialog* and *TWindow* descend from *TWindowsObject*.
- Dialog boxes normally require *resources* that specify their size, location, and appearance.
- Dialog boxes usually perform a short task and return a value. For example, the *CanClose* message box, a limited type of dialog you created in Step 2 returned a yes or no answer from the user.

You'll add one of ObjectWindows' stock dialog boxes, the file dialog box, defined by type *TFileDialog*, a descendant of *TDialog*. The file dialog box is useful in any situation where you are asking 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 will replace the "Feature not implemented" message box. In Step 9, you'll hook it up to some real files and save and open them to store and retrieve real data. For now, you'll just show the dialog boxes. Figure 5.3 shows the appearance of the file dialog box.

Figure 5.3
MyProgram with the file dialog box



Adding an object field

Instead of storing an entire file dialog box object as a field of its parent window, you'll construct a new file dialog box object each time you need one. What you'll 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 *TMyWindow* type definition should now look like this:

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

Modifying the Init constructor

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

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

The .PTS extension will be used on files that hold 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 5.3. 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 you'll rewrite *FileOpen* and *FileSaveAs*:

```
procedure TMyWindow.FileOpen(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 TMyWindow.FileSaveAs(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;
```

The full source code to *MyProgram*, to this point, is in the file STEP8.PAS.

Just as you used *TApplication.MakeWindow* to create "safe" interface elements, you can use *TApplication.ExecDialog* to create

and execute safe dialog boxes. Like *MakeWindow*, *ExecDialog* calls *ValidWindow* to make sure that constructing the dialog box object was successful. If it was, then *ExecDialog* calls the dialog box object's *Execute* method. After the dialog box executes, *ExecDialog* returns the value returned by *Execute*, unless an error occurred, in which case it returns *id_Cancel*, so it looks to your program as if the user cancelled the dialog box. In any case, *ExecDialog* dissposes the dialog box object.

Using *ExecDialog* is the safest way to run dialog boxes. You can call *Execute* directly, but doing so is dangerous, and you risk hanging your system.

Step 9: Storing the drawing in a file

Now that you've got a data representation of the drawing stored as part of the window object, you should be able to transfer that data into a file (actually, a 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.

Monitoring the status

There are two characteristics of the drawing you need to monitor. The first is whether the file needs saving and the second is whether there is a file currently loaded. You can think of these characteristics as Boolean attributes of *TMyWindow*, so make them object fields:

```
TMyWindow = object(TWindow)
  ...
  IsDirty, IsNewFile: Boolean;
  ...
end;
```

IsDirty is *True* if the current drawing is "dirty." By dirty, you mean that it needs to be saved because it has changed since it was last saved, or because it has never been saved. When the user starts drawing (*WMLButtonDown*), you should set *IsDirty* to *True*. When the user opens a new file or saves the existing one, you should set *IsDirty* to *False*. When the user closes the application (*CanClose*), you should check the status of *IsDirty* and display the message box only if it is *True*.

IsNewFile is *True* only when the application first starts (*Init*) and after the user selects the File | New menu (*FileNew*). It is set to *False* whenever a file is opened (*FileOpen*) or saved (*FileSave* or *FileSaveAs*). In fact, *FileSave* uses *IsNewFile* to see if the file can be saved immediately (*False*) or if the user needs to select a file from a file dialog (*True*).

Listed here are the *CanClose* method and the file saving and 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 *SaveFile*, and file opening is delegated to *LoadFile*.

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

procedure TMyWindow.FileNew(var Msg: TMessage);
begin
  if CanClose then
  begin
    Points^.FreeAll;
    InvalidateRect(HWindow, nil, True);
    IsDirty := False;
    IsNewFile := True;
  end;
end;

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

procedure TMyWindow.FileSave(var Msg: TMessage);
begin
  if IsNewFile then FileSaveAs(Msg) else SaveFile;
end;
```

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

procedure TMyWindow.LoadFile;
begin
  MessageBox(HWindow, @FileName, 'Load the file:', mb_OK);
  IsDirty := False;
  IsNewFile := False;
end;

procedure TMyWindow.SaveFile;
begin
  MessageBox(HWindow, @FileName, 'Save the file:', mb_OK);
  IsNewFile := False;
  IsDirty := False;
end;
```

Saving and loading Files

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'll have to teach points to store and load themselves (since collections already know how). Then you'll modify the *SaveFile* and *FileOpen* methods to make use of streams.

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

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

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

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

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

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

Then, you must call *StreamRegistration* when the application starts up. You can put this call in the *TMyWindow.Init* method.

The final step is to rewrite the *SaveFile* and *LoadFile* methods:

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

procedure TMyWindow.SaveFile;
var
  TheFile: TDosStream;
begin
  TheFile.Init(FileName, stCreate);
  TheFile.Put(Points);
  TheFile.Done;
  IsNewFile := False;
  IsDirty := False;
end;
```

C H A P T E R

6

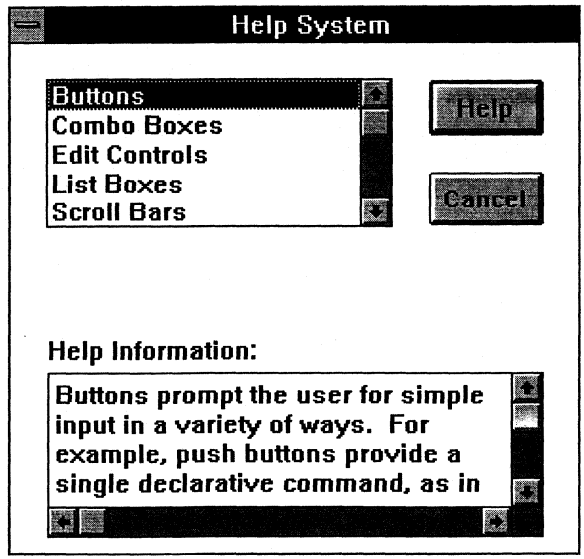
Popping up windows

Now you're ready to add the finishing touch to your ObjectWindows program, the help system. You can think of a help system as a program within a program that can be invoked at a moment's notice. One of the benefits of a Windows program is that, with its multiple windows, it can encompass and unify many activities. Adding a help system to *MyProgram* will illustrate how to design ObjectWindows programs in a modular format, using units, for a high degree of reuse and maintainability.

Step 10: Popping up a help window

In Step 8, you added a Help menu to *MyProgram*'s main window. Selecting this menu produced a plain window, an object instance of type *TWindow*. In this part, you'll replace that plain window with a useful help window that makes use of some of the Windows controls supported by ObjectWindows: list boxes, scroll bars, edit controls, static controls, buttons, and combo boxes. In the process, you will define the help system in a unit, requiring little change in your existing program. Figure 6.1 shows the finished help system, which provides a small description of each of the controls ObjectWindows supports.

Figure 6.1
MyProgram's help system



Using units with ObjectWindows

A Turbo Pascal unit is a convenient storage mechanism for definitions of objects or groups of related objects. This use is especially important in Windows programming, where you tend to reuse windows (window objects in ObjectWindows) from one application to the next. Here, you'll build your help system as an object type called *THelpWindow* and store its definition in a unit called *HelpWind*.

Modifying the main program

MyProgram requires only two changes:

- Add the *HelpWind* unit to the **uses** statement.
- Remove all of the code to set up the attributes of *HelpWind* from *TMyWindow.Help*. This code will now go into *THelpWindow.Init*.

Now, when the user selects the Help menu item, *MyProgram* responds by producing a *THelpWindow*. That means you need to define the *THelpWindow* type. You'll do that in the *HelpWind* unit.

Creating the unit

The *HelpWind* unit will do nothing more than define the interface and implementation of the type *THelpWindow*. Once created, this unit can be used by any other ObjectWindows program. It is, in effect, a window in a unit. Here is the format of the unit (the gaps will be filled in later):

```
unit HelpWind;

Interface

uses Strings, WinTypes, WinProcs, WObjects;

const
  id_LB1 = 201;
  id_BN1 = 202;
  id_BN2 = 203;
  id_EC1 = 204;
  id_ST1 = 205;

type
  PHelpWindow = ^THelpWindow;
  THelpWindow = object(TWindow)
    LB1: PListBox;
    EC1: PEdit;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    procedure SetupWindow; virtual;
    procedure IDLB1(var Msg: TMessage); virtual id_First + id_LB1;
    procedure IDBN1(var Msg: TMessage); virtual id_First + id_BN1;
    procedure IDBN2(var Msg: TMessage); virtual id_First + id_BN2;
    procedure FillEdit(SelStringPtr: PChar); virtual;
end;

Implementation

constructor THelpWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  ...
end;

procedure THelpWindow.SetupWindow;
begin
  ...
end;

procedure THelpWindow.IDLB1(var Msg: TMessage);
var
  ...
begin
  ...
```

```
end;

procedure THelpWindow.IDBN1(var Msg: TMessage);
var
  ...
begin
  ...
end;

procedure THelpWindow.IDBN2(var Msg: TMessage);
begin
  ...
end;

procedure THelpWindow.FillEdit(SelStringPtr: PChar);
var
  ...
begin
  ...
end;

end.
```

Once an ObjectWindows application lists *HelpWind* in its **uses** statement, it can instantiate type *THelpWindow* by defining variables to be the *PHelpWindow* type.

Adding controls to a window

In Step 8, you learned that there are two types of child windows: dependent and independent. In that part you added to the main window two types of independent child windows: dialogs and popup windows. Now, you'll add a type of dependent child window called a *control*. These controls will have the help window as a parent window. Remember that the help window is an independent child window whose parent window is your application's main window. Therefore, your help window is simultaneously a child window and a parent window. This is allowed, and required, for a Windows application with any complexity.

What are controls?

Controls are visual devices that make up part or all of a window's or dialog boxes' user interface. For example, the Save and Open buttons on file dialog boxes are controls, as are the combo boxes

you used to display files. *MyProgram*'s main window has no controls, but here you'll add some to your help window.

There are many types of controls, including list boxes, scroll bars, edit controls, static controls, buttons, and combo boxes. Your help window, pictured in Figure 6-1, makes use of a list box (the large box in the top half of the window), an edit control (the large box in the lower half of the window), a static control (the text "Help Information:"), and two buttons (Help and Cancel). Scroll bars are part of both the edit control and the list box, but these are not scroll bar objects. They are most often used as optional parts of other controls and windows and rarely appear as standalone controls.

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*. Controls are actually specialized windows.

Creating window controls

While they behave identically, there's an important programming difference between the controls in dialog boxes, such as file dialogs, and the controls in windows, such as in your help window. You specify the controls of a dialog box in the dialog's resource. They are not objects and the dialog boxes that own them are completely responsible for managing them. Chapter 5 shows how to create your own dialogs from dialog resources and to handle their controls.

Dialog controls can have objects associated with them as well. See "Associating control objects" in Chapter 11.

A window's controls, however, are specified by object definitions. A parent window manages its controls through the rich set of 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_LB1 = 201;                          { ID for list box 1 }
  id_BN1 = 202;                      { ID for first (OK) button }
  id_BN2 = 203;                  { ID for second (Cancel) button }
  id_EC1 = 204;                          { ID for edit control }
  id_ST1 = 205;                          { ID for static text }
```

Control objects as fields

It is sometimes convenient to store a pointer to a control object (or other child window) 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, only the list box and edit control qualify. *THelpWindow* will store each of these control objects in a separate field. Here is part of *THelpWindow*'s object definition:

```
THelpWindow = object(TWindow)
  LB1: PListBox;
  EC1: PEdit;
  ...
end;
```

One these child control objects have been instantiated, you can manipulate them with method calls. For example, you can add a string to the list in *LB1* by calling *LB1^.AddString*. 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 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 (*THelpWindow*) automatically creates and displays all of its child windows.

Listed here is the help window'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. A control ID is the second parameter in every control's constructor call.

```
constructor THelpWindow.Init(AParent: PWindowsObject; ATitle: PChar);
var
  TempStat: PStatic;
  TempBtn: PButton;
begin
  TWindow.Init(AParent, ATitle);
  DisableAutoCreate;
  Attr.Style := ws_PopupWindow or ws_Caption or ws_Visible;
  Attr.X := 100;
  Attr.Y := 100;
  Attr.W := 300;
  Attr.H := 300;
  LB1 := New(PListBox, Init(@Self, id_LB1, 20, 20, 180, 80));
  TempBtn := New(PButton, Init(@Self, id_BN1, 'Help', 220, 20, 60,
    30, True));
  TempBtn := New(PButton, Init(@Self, id_BN2, 'Cancel', 220, 70, 60,
    30, False));
  EC1 := New(PEdit, Init(@Self, id_EC1, 20, 180, 260, 90, 0, True));
  EC1^.Attr.Style := EC1^Attr.Style or ws_Border or ws_VScroll;
  TempStat := New(PStatic, Init(@Self, id_ST1, 'Help Information:',
    20, 160, 160, 20, 0));
end;
```

THelpWindow.SetupWindow is called following the window's creation in order to set up the window's controls. You will use *TListBox*'s modification methods, such as *AddString*, to add items to the list box. Whenever you override a window's *SetupWindow* method, be sure to call *TWindow.SetupWindow* first, because it creates all of the child controls.

```
procedure THelpWindow.SetupWindow;
begin
  TWindow.SetupWindow;
  { Fill the list box }
  LB1^.AddString('List Boxes');
  LB1^.AddString('Buttons');
  LB1^.AddString('Scroll Bars');
  LB1^.AddString('Edit Controls');
  LB1^.AddString('Static Controls');
  LB1^.AddString('Combo Boxes');
  LB1^.SetSelIndex(0);
end;
```

Init and *SetupWindow* methods are enough to get all the controls displayed properly in the help window. The list box will scroll and buttons will press, but with no resulting action. In the next section you'll define a response to control events.

Responding to control events

At this point, the controls will appear in the help window, but clicking on the buttons and selecting list box items will not have any effect. That's because clicking and selecting are control events. They are very similar to the menu events you responded to in Step 7.

You responded to menu events by defining command message response methods, and you'll do something similar with control messages. However, 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. For clarity, give these message response methods names like *IDLB1*. Here's the *THelpWindow* type definition showing the method headers:

```
THelpWindow = object(TWindow)
  LB1: PListBox;
  EC1: PEdit;
  constructor Init(AParent: PWindowsObject; ATitle: PChar);
  procedure SetupWindow; virtual;
  procedure IDLB1(var Msg: TMessage); virtual id_First + id_LB1;
  procedure IDBN1(var Msg: TMessage); virtual id_First + id_BN1;
  procedure IDBN2(var Msg: TMessage); virtual id_First + id_BN2;
  procedure FillEdit(SelStringPtr: PChar); virtual;
end;
```

Next, decide how *THelpWindow* should respond to control events. When the user clicks on the Cancel (BN2) button, the window should close. Thus the implementation of *IDBN2* is the most simple:

```
procedure THelpWindow.IDBN2(var Msg: TMessage);
begin
  Destroy;
end;
```

However, when the user clicks on the OK (BN1) button, you want *MyProgram* to check to see which list box item is selected and fill up the edit control (EC1) with corresponding text. Thus, *IDBN1* calls *GetSelString* for the list box (LB1) to get the text of the selected list item. The text is passed to a method you must write called *FillEdit*. *FillEdit*'s definition is listed in the full code listing at the end of this step.

```
procedure THelpWindow.IDBN1(var Msg: TMessage);
var
  SelString: array[0..25] of Char;
begin
  LB1^.GetSelString(SelString, 25);
  FillEdit(SelString);
end;
```

As an added touch, you can respond to the message generated when the user double-clicks directly on a list box item by also filling the edit control with text. However, a list box can generate a variety of messages (called list box notification messages) in response to clicking, double-clicking, scrolling and more. To distinguish between all the messages, you define *IDLB1* to check *Msg.LParamHi*, where the list box notification is held. If the value is *lbn_DblClk*, the user double-clicked. You will ignore all other codes, which will then be handled by the default window procedure, *DefWndProc*.

```
procedure THelpWindow.IDLB1(var Msg: TMessage);
var
  SelString: array[0..25] of Char;
begin
  if Msg.LParamHi = lbn_DblClk then
  begin
    LB1^.GetSelString(SelString, 25);
    FillEdit(SelString);
  end
  else DefWndProc(Msg);
end;
```

ObjectWindows's control objects provide a variety of querying and modifying methods. Calling these methods in response to control events will animate your windows, as you have done with the help window.

PART 2

Using ObjectWindows

C H A P T E R

7

Object Windows overview

ObjectWindows, the windows object library, is a comprehensive set of object types that will streamline your development of Microsoft Windows programs with Turbo Pascal.

This chapter gives an overview of the ObjectWindows object hierarchy. The remaining chapters in this part provide detailed descriptions of the 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 use of resources, calling the Windows API, and receiving and handling messages from Windows.

ObjectWindows conventions

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

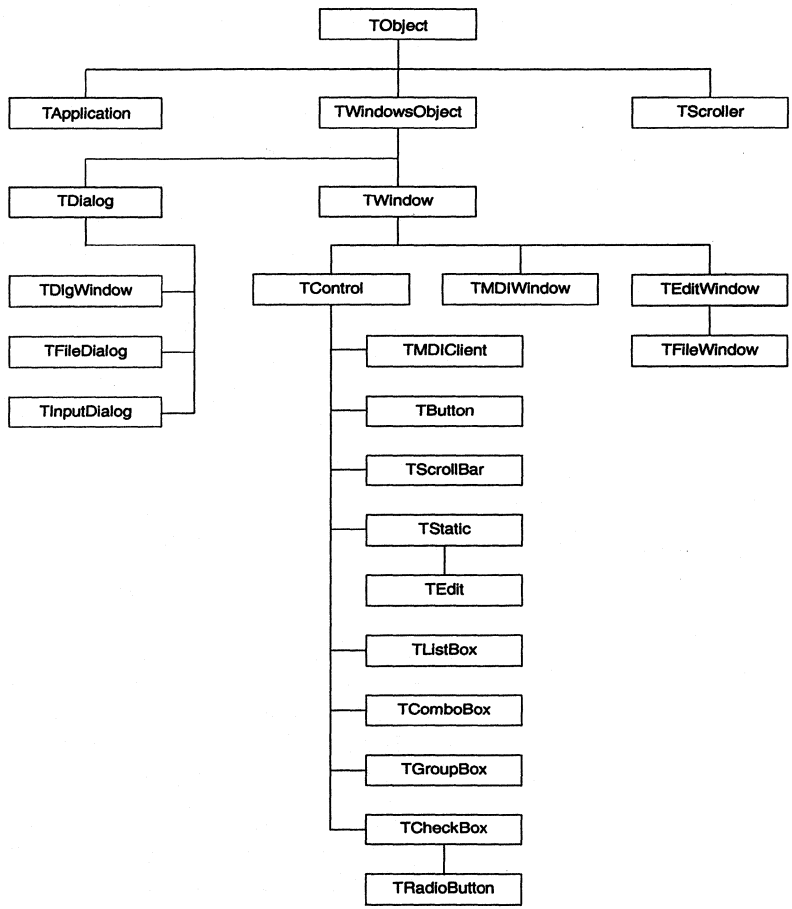
Message-response methods are named after the messages they respond to, but without the underscores. For example, a method responding to the *wm_KeyDown* message would be called

WMKeyDown, and a message responding to the *cm_FileOpen* command would be called *CMFileOpen*.

The ObjectWindows hierarchy

ObjectWindows is a library consisting of a hierarchy of object types that you can use, modify, or add to, using inheritance. Chapter 5 in the *Windows Reference Guide* is a complete reference to these types and their fields and methods. This manual is a complete user's guide to the library's object types and methods. It also provides sample programs to guide you through the steps involved in building Windows programs with ObjectWindows.

Figure 7.1
ObjectWindows object type hierarchy



Base objects

TObject is the base object type, the common ancestor of all ObjectWindows object types. 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 that descends from *TApplication*. It is responsible for, among other things, initialization of the main window object. 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 both in the sense that they represent elements in the Windows user interface, and because they serve as a kind of interface between your application code and the Windows environment. Interface objects are described in detail in Chapter 9, "Interface objects."

TWindowsObject

TWindowsObject is an abstract object type that unifies the three main types of ObjectWindows interface objects: windows, dialog boxes and controls. It provides methods to handle the creation, message processing, and destruction of window 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.

TWindow

TWindow objects are general-purpose windows that can serve as the main window of an application, or as one of its pop-up windows. Object instances of *TWindow* can display graphics, but most often you will specialize *TWindow*'s behavior by defining descendant types.

TEdit Window — A descendant of *TWindow*, *TEditWindow* defines a window that allows text editing.

TFileWindow — A descendant of *TEditWindow*, *TFileWindow* defines a window that loads, edits, and saves text files.

Dialog objects

Dialog objects serve to facilitate interactive groups, particularly groups of controls such as buttons, list boxes, and scroll bars. Dialog objects are explained in detail in Chapter 11, "Dialog objects."

TDialog — This abstract object type serves as a template for descendant types that manage Windows dialog boxes. Dialog objects are associated with dialog resources, and can be run as either modal or modeless dialog boxes. Methods are provided to handle communication between a dialog and its controls.

TDlgWindow — This type combines the resource specification of dialog objects with the customizability of window objects.

TFileDialog — *TFileDialog* is a stock dialog type that is immediately useful in many applications. It defines a dialog that allows the user to choose a file for any purpose, such as opening, editing, or saving.

TInputDialog — This type defines a stock dialog box for user input of a single text item.

Control objects

Within dialogs 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."

TControl — *TControl* is an abstract object type that serves as a common ancestor type for every variety of Windows controls, including list boxes and buttons. It defines methods that create controls and process messages for its descendant types.

TButton — *TButton* handles the creation of Windows push buttons.

TListBox — This class handles creation of and selection from Windows list boxes.

TComboBox — A descendant type of *TListBox*, *TComboBox* defines behavior for Windows combo boxes. A combo box is an interdependent list box and edit control.

TCheckBox — *TCheckBox* handles creation and state management for Windows check boxes.

TRadioButton — *TRadioButton* handles creation and state management for Windows radio buttons.

TGroupBox — *TGroupBox* handles creation of Windows group boxes.

TEdit — *TEdit* supports the extensive text processing capabilities for a Windows edit control.

TScrollBar — This type supports the creation and management of a standalone scroll bar control.

TStatic — *TStatic* provides methods which set, query, and clear the text of a static (output only) control.

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 13, "MDI objects."

TMDIWindow *TMDIWindow* provides the windowing behavior appropriate for the main window of an application that follows the Windows MDI specification.

TMDIClient *TMDIClient* provides additional support for MDI windows. The MDI client object is the object that actually manages the MDI window's client area, the area which contains the child windows ("documents").

ObjectWindows units

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

WObjects

WObjects must be used by all ObjectWindows applications. It defines all the standard types for windows objects, abstract objects like *TObject*, and other useful types which implement collections and streams.

WinProcs

WinProcs defines function and procedure headers for the Windows Applications Programming Interface (API). Every routine provided by the standard Windows libraries can be accessed through *WinProcs*. Together with *WinTypes*, *WinProcs* defines the Turbo Pascal implementation of the Windows API. The *WinProcs* unit and its contents are described in the "Windows functions" section on page 85 of this chapter.

WinTypes

WinTypes defines Turbo Pascal versions of all the types used by Windows API routines, including simple types and data structures (records), and all the standard Windows constants, including styles, messages, and flags.

StdDlgs

StdDlgs defines two standard dialog types, *TFileDialog* and *TInputDialog*.

StdWnds

StdWnds defines two standard window types, *TDlgWindow* and *TFileWindow*.

Windows resources

ObjectWindows includes two resource files: STDDLGS.RES, which holds the resources required by *TFileDialog* and *TInputDialog* objects, and STDWNDS.RES, which holds the resources used by the *TFileWindow* and *TEditWindow* objects. The resources are automatically included when the corresponding units are used, so any program that uses the *StdDlgs* unit will automatically have access to the resources in STDDLGS.RES.

Windows 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 Windows functions

The methods of ObjectWindows call Windows functions. But ObjectWindows isn't duplicating functionality; it's repackaging Windows' list of functions, its application program interface (API), into a dynamic, 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 functions

In order to directly access any of the Windows functions from your ObjectWindows applications, you must use the *WinProcs* unit. *WinProcs* defines a Turbo Pascal procedure or function header for each Windows function, allowing you to call Windows functions just as you would any Turbo Pascal routine. See Chapter 2, "Windows function reference," in the *Windows Reference Guide* or the file WINPROCS.INT for a listing of the function headers.

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 which uses numbers. For example, it is more informative to define a window with the style *ws_Popup*, rather than $80000000. *ws_Popup* and the other constants are defined in the *WinTypes* unit, and are described in Chapter 1, "Windows styles and constants," in the *Windows Reference Guide.*

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 Chapter 4, "Windows type reference," in the *Windows Reference Guide*, or the file WINTYPES.INT.

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 on the Yes button, the result is equal to the Windows-defined integer constant *id_Yes*. If the user clicked on the No button, the result is *id_No*.

Combining style constants

Windows functions that produce interface elements usually require some style parameter of type *Word* or *Longint*. Windows defines hundreds of style constants, which are listed in Chapter 1, "Windows styles and constants," in the *Windows Reference Guide*. 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*.

You can pass a pointer to a virtual method as the callback function in these methods, but that method will always be called, regardless of polymorphism. In other words, the address is resolved at compile time, so that particular method will always be the one called. In most cases, a static method or regular function should be used. 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;
```

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.

Receiving Windows messages

A typical Windows application is little more than a message dispatcher and message processor. For example, when the user clicks the mouse in a window, Windows sends a message to the application. Like Windows functions, there are many different Windows messages, each with its own name, like *wm_LButtonDown* and *wm_Command* (*wm_* stands for Windows Message). A Windows message is like a function call in that it carries parameters filled with important data. In a way, a message is Windows' way of calling a function in your program.

An application is responsible for intercepting the messages and dispatching them to the appropriate part of the application's code. For example, a Windows application might define a routine to handle the case where the user clicked the mouse in a window. ObjectWindows dispatches the messages automatically, with help of an addition to the Turbo Pascal virtual method table (VMT) called a dynamic method table (DMT). The DMT allows you to extend a method definition to include an identifier which corresponds to a Windows message. All you have to do is write methods that respond to Windows messages. We call these methods *response methods.*

Here is a response method definition header for a method which is called when the user clicks the left mouse button:

```
procedure WMLButtonDown(var Msg: TMessage); virtual wm_First +
  wm_LButtonDown;
```

How events become method calls

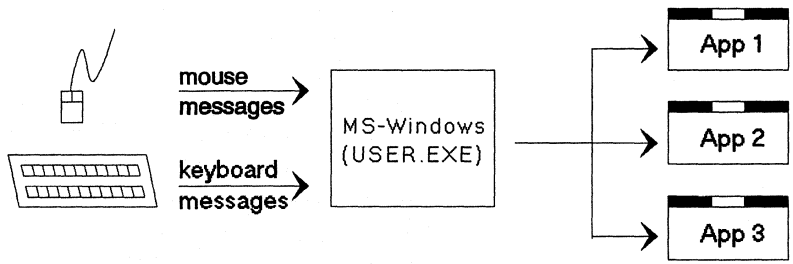
To become a successful ObjectWindows programmer, you need to learn how to write response methods. In order to do that, however, you should understand what types of events result in what types of messages. Windows messages can come from these sources:

- The user, by moving or clicking the mouse or typing on the keyboard, generates a user-event message.
- Your program can send messages directly to itself.

- Your program can call Windows functions which result in its receiving other Windows messages.
- Applications can send messages to other applications through dynamic data exchange (DDE).
- The Windows shell, including the Program Manager, sends a message to all open applications before it closes.

Responding to user-event messages is the primary means for interacting with the user. This is *event-driven programming*. When an event occurs, the Windows module, USER.EXE, generates a message and places it in an application's *message queue*. While there is only one instance of USER.EXE for a Windows session, each currently running application has its own message queue. Each application is responsible for polling its message queue and responding to its messages. ObjectWindows takes care of this polling for its applications. This is how Windows' multitasking works. See Figure 7.2.

Figure 7.2
How a message gets to an application



Inside each Windows application is a piece of code called a *message loop*, which runs during the course of the program's operation. The message loop polls the message queue and dispatches the message throughout the program. The message loop of an ObjectWindows application is the *MessageLoop* method of its *TApplication* object.

Each message from Windows identifies the window which is most associated with the instigating event by including the handle of the window in the message record. ObjectWindows dispatches the message to the window object associated with that window handle. ObjectWindows determines whether that window defines a response method for that particular message, and if so, calls that method. Otherwise, the default response is invoked. This guarantees that all messages are responded to in *some* way.

For example, if the user clicked the mouse on a window, the resulting Windows message would carry a handle to that window. As a result, a window object receives messages for which its type may or may not define response methods. If it does not define a response method, ObjectWindows invokes the default message response, the object's *DefWndProc* method. Default message processing is described in Chapter 9, "Interface objects."

The parameters of a Windows message

Each Windows message comes along with a handle to a message structure with six fields. Two are used internally by Windows. Another two, the window handle and the message identifier, are used by ObjectWindows to properly dispatch the correct message to the correct window. Only two parameters hold data directly useful to your program.

wParam is a *Word* that holds one piece of information, usually a handle, identifier, or code. For example, a menu selection generates a *wm_Command* message whose *wParam* equals the ID of the menu item selected.

lParam is a *Longint* that usually holds two pieces of information, one in the low-order *Word* and one in the high-order *Word*. *lParam* usually holds more complex data such as coordinate points or pointers to strings (*PChar*). For example, the *wm_LButtonDown* message, generated when the user clicks the left mouse button in a window, carries the x-coordinate of the clicked point in the low-order word of *lParam* and the y-coordinate in the high-order word. Because *TMessage* is defined as a variant record, the low- and high-order words can be accessed as *lParamLo* and *lParamHi*, respectively.

Types of Windows messages

Listed below are the types of messages a Windows application can receive.

Window-management messages

These messages signal the state of a window has changed. For example, *wm_Close* is sent when a window is closed and *wm_Paint* is sent when part or all of a window needs to be re-displayed.

Initialization messages These messages are sent when a program creates a menu (*wm_InitMenu*) or dialog box (*wm_InitDialog*).

Input messages These messages are received as a result of input through the mouse (*wm_MouseMove*), keyboard (*wm_Char*), scroll bars (*wm_VScroll*), or system timer (*wm_Timer*). One of the most common input messages is *wm_Command*, which results from a menu or accelerator selection, or from a control event, such as clicking on a button.

System messages These messages are sent in response to the user's manipulation of an application's standard accessories, such as its Control-menu box, scroll bars, or minimize or maximize button. *wm_SysCommand* is sent as a result of a Control-menu box selection. Most applications do not intercept system messages.

Clipboard messages These messages are sent as a result of another application's attempt to access your window's Clipboard.

System-information messages These messages are sent when a Windows system-level attribute, such as color (*wm_SysColorChange*) or font (*wm_FontChange*), changes.

Control-manipulation messages These messages are sent by an application to its controls, such as list boxes and buttons. Each control type understands its own set of messages for querying and setting its attributes. Control messages differ from all other messages except MDI messages in that they do not *notify* the application about an event, they *cause* an event. See "Sending messages from within your program" on page 93.

Control-notification messages These are *wm_Command* messages that notify a parent window of a control event, such as a list box item being selected or an edit control being typed into. The message's parameters specify a particular notification code, such as *lbn_SelChange* and *en_Change*.

Scroll-bar-notification messages

These are specialized control-notification messages for scroll bar controls. There are two messages, *wm_HScroll* and *wm_VScroll.*

Non-client area messages

These messages are similar to input messages, but deal specifically with events outside a window's client area, including its menu bar, title bar, and borders. They include *wm_NCMouseMove* and *wm_NCPaint*. Usually your application will not need to use these messages.

Multiple document interface (MDI) messages

These messages are sent by an application that follows the MDI standard to manipulate its child windows. MDI messages include *wm_MDIActivate* and *wm_MDIDestroy.*

Default message processing

Some Windows messages *require* a particular default action or return value. The Windows function *DefWindowProc* provides the default actions for each message. ObjectWindows calls *DefWindowProc* automatically for any ignored messages. However, intercepting these messages with response methods might prevent required actions. This is true only in some Windows messages, and in these cases, you should call *DefWindowProc* directly (through the *TWindowsObject* method *DefWndProc*).

Sending messages

One source of messages is your program itself. Your program can call a Windows function, *SendMessage.* As a result, your program receives a message. This technique is also useful to simulate an event from within your program.

SendMessage is most useful when manipulating controls, such as list boxes and buttons. Windows defines *control-manipulation messages* to do such things as add items to a list box (*lb_AddString*) or change the state of a check box (*bm_SetCheck*). ObjectWindows's control objects define methods such as *TListBox.AddString* and *TCheckBox.SetCheck* that send many of these messages (those commonly used) to their associated interface elements. If you want to send some of the more obscure messages to controls in your windows, use *SendMessage,* using the control's *HWindow* as the first (*Wnd*) parameter.

When you want to send a message to a control in a dialog box, the procedure is somewhat different. Often, the controls in your *TDialog* objects will not have their own interface objects associated with them, but you can send messages to them using *TDialog*'s *SendDlgItemMsg* method. You can facilitate communication with controls in your dialog boxes by associating interface objects with them, using the *InitResource* constructor. See "Associating control objects" in Chapter 11.

Message ranges

Since messages are defined by *Word*-type values, there are 65,536 possible messages. Windows divides messages into unique ranges for standard Windows messages, command messages, notification messages, and so on. To make these ranges more identifiable, ObjectWindows defines constants to use as offsets for the separate ranges. The message ranges and their ObjectWindows constants are listed in Table 7.1.

Table 7.1
Ranges of Windows messages

Constant	Value	Range	Meaning
wm_First	$0000	$0000-$7FFF	Window messages
wm_User	$0400	$0400-$8FFF	User-definable messages
id_First	$8000	$8000-$8FFF	Child-ID messages
id_Internal	$8F00	$8F00-$8FFF	Reserved for internal use
nf_First	$9000	$9000-$9FFF	Notification messages
nf_Internal	$9F00	$9F00-$9FFF	Reserved for internal use
cm_First	$A000	$A000-$FFFF	Command messages
cm_Internal	$FF00	$FF00-$FFFF	Reserved for internal use

User-defined messages

Windows allows you to define your own messages for use by your application. User-defined messages are useful when defining and responding to a new event. For example, you can have one window send a user-defined message to all the remaining open windows in the application.

Associated with each Windows message is a *Word*-type identifier. For each of the predefined Windows messages, ObjectWindows defines a corresponding *Word*-type constant, which is the constant used in the method header of a message response method.

Within the allowable range of Windows message identifiers are values reserved for user-defined messages. The range of values reserved for predefined Windows messages is 0 to *wm_User* – 1. (*wm_User* is a constant.) The allowable range of values for user-defined messages is *wm_User* to *wm_User* + 31,744. It's probably best to define user-message values as constants, starting with the values *wm_User*, *wm_User* + 1, *wm_User* + 2, and so on. For example,

```
const
  wm_MyFirstMessage  = wm_User;
  wm_MySecondMessage = wm_User + 1;
  wm_MyThirdMessage  = wm_User + 2;
```

Then send your messages using the Windows function, *SendMessage*:

```
SendMessage(AWindow^.HWindow, wm_MyFirstMessage, AWord, ALongint);
```

To receive a user-defined message, follow the same procedure as for a standard Windows message. For example,

```
MyWindow = object(TWindow)
  ...
  procedure WMMyFirstMessage(var Msg: TMessage); virtual wm_First +
    wm_MyFirstMessage;
  procedure WMMySecondMessage(var Msg: TMessage); virtual wm_First +
    wm_MySecondMessage;
  procedure WMMyThirdMessage(var Msg: TMessage); virtual wm_First +
    wm_MyThirdMessage;
end;
```

C H A P T E R

8

Application objects

The first requirement of an ObjectWindows application is the definition of an application object derived from the abstract type, *TApplication*. The application object will encapsulate the following behavior of an ObjectWindows application:

- Creating and displaying the application's main window.
- Initializing every instance of an application, for example, to load an accelerator table. (You *can* simultaneously run many instances of the same ObjectWindows application.)
- Initializing the first instance of an application for any tasks that serve all instances of the application, such as configuring a communications port.
- Processing the Windows messages an application will receive.
- Closing the application.

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

Controlling an application's flow

Here's a minimal ObjectWindows application's main program:

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

This program, *MinimalApp*, is the absolute minimum ObjectWindows application, and it requires no definition of new object types. Typically, however, you will define new object types for at least the application and the main window.

The application object is constructed in an ObjectWindows application's main program. If needed, it can be accessed from other parts of an ObjectWindows program by referring to the global variable, *Application*, in which it is also stored.

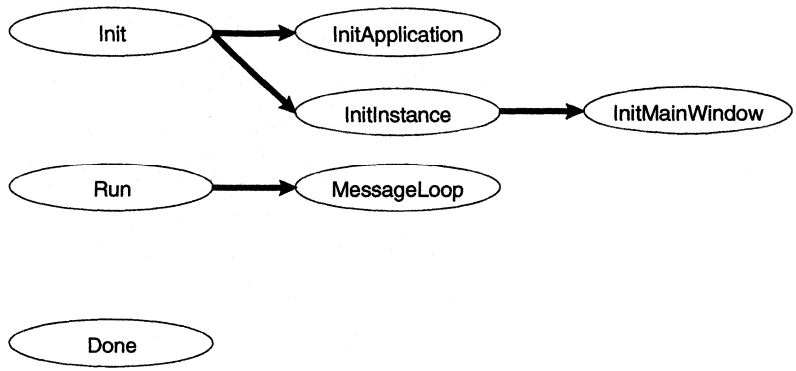
The main program of your ObjectWindows application will normally consist of just three statements.

The first statement constructs the application object by calling its *Init* constructor. *Init* also initializes the data fields of the application object, then calls *InitApplication* and *InitInstance*, which perform first-instance and each-instance initialization, respectively. *InitMainWindow* is then called to create a main window. When *Init* has finished, the main window of your application is on the screen. In most cases, you will need to redefine only *InitMainWindow* (see Figure 8.1).

The second statement calls the application's *Run* method, which sets the application in motion by calling *MessageLoop* to begin processing incoming Windows messages, the instructions that direct an application's activity. *MessageLoop* calls methods that process the particular incoming messages. *MessageLoop* is exactly that, a message loop, which continues running until the application closes.

Done is the destructor for application objects that frees the application object's memory prior to application termination.

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 many parts of the process by redefining methods. The one required method you must write is *InitMainWindow*. You can also redefine *InitInstance* to initialize each executing instance of an application, and *InitApplication* to initialize the first executing instance of a ObjectWindows application.

Initializing the main window

Here is a minimal application object type definition:

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

You must define an *InitMainWindow* method, which constructs and initializes the main window object and stores it in the application object's *MainWindow* field. Here is a simple *InitMainWindow* method:

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

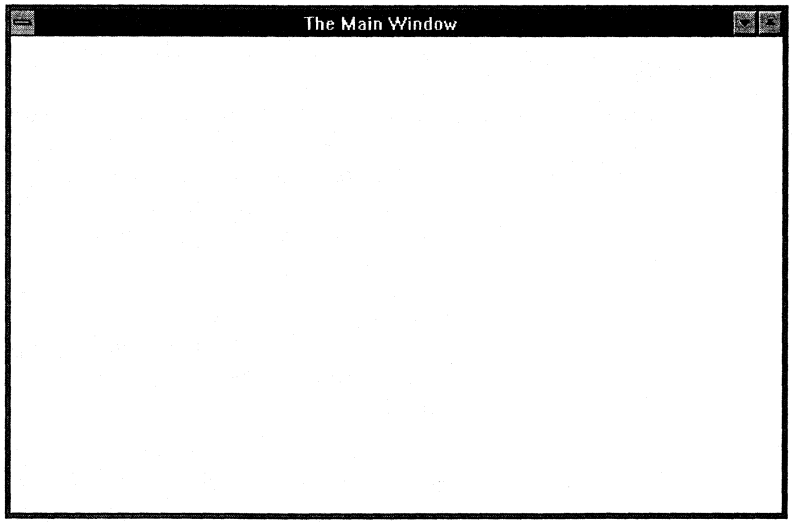
This method creates a new object instance of the ObjectWindows type *TWindow* (*PWindow* is a pointer to the type *TWindow*). Normally, your program will define a new window type for its main window, and your *InitMainWindow* will use that new type, rather than *TWindow*.

This simple ObjectWindows application consists of a main program plus an application type definition that defines one method, *InitMainWindow*. Here is the source code for the program, which differs from the earlier *MinimalApp* only in that its main window has a caption:

```
program TestApp;

uses WObjects;

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

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

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

TestApp is a minimal application that simply displays a window with the caption 'The Main Window', as shown in Figure 8.2. You can move and resize this window. You can also minimize it to an icon by clicking on the ↓ icon in the upper right corner. To restore it, double-click on the icon. Clicking on the ↑ icon maximizes it to take over the full screen. Close the window, and terminate the application, by double-clicking on the Control-menu box in the upper left corner. *TestApp* is a fully-functional, bare-bones application featuring only the simplest main window.

Figure 8.2
The Main Window



Initializing each application instance

The user can execute multiple instances of one ObjectWindows application simultaneously. The *InitInstance* method initializes each instance of the application. It should only perform initialization for the application instance. Main window initialization should be done in *InitMainWindow*. *TApplication* defines an *InitInstance* method which calls *InitMainWindow* and then displays the main window with a call to its *Show* method. You need only redefine *InitInstance* to modify the standard initialization, for example, to load an accelerator table, an application-oriented activity. If you redefine *InitInstance* for your application type, be sure it calls *TApplication.InitInstance* first.

Here is an *InitInstance* method that loads an accelerator table before the application is set in motion. 100 is the constant identifier of the accelerator table, defined in the resource file.

```
procedure TEditApplication.InitInstance;
begin
  TApplication.InitInstance;
  HAccTable := LoadAccelerators(HInstance, PChar(100));
end;
```

Initializing the first application instance

If the user runs your application many times, simultaneously (without terminating it), you can define some processing to occur the first time it is run. We will call this processing the first instance initialization. Note that if the user starts and terminates your application, starts it again, and so on, each instance is considered a first instance.

If the current instance is a first instance, its *InitApplication* method will be called. *TApplication* defines a placeholder *InitApplication* method that can be redefined to perform special first instance initialization.

For example, you can modify *TestApp* to make the main window's caption reflect whether or not it is the first instance. To do this, add a Boolean object field called *FirstApp* to the application type, *MyApplication*. 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 8.3.

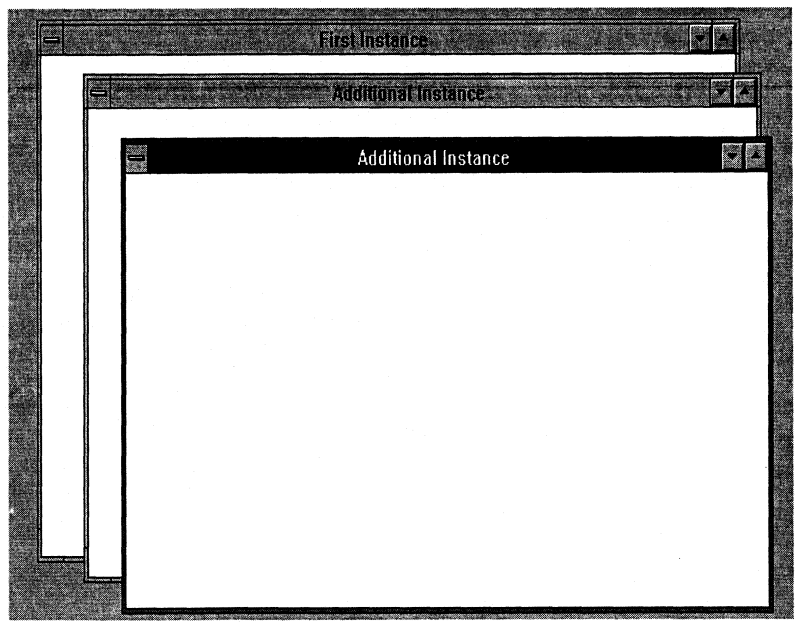


Figure 8.3
Refining application initialization

```
program TestApp;

uses WObjects;

type

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

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

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

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

Running applications

Your application's message loop is invoked when your program calls its application object's *Run* method, which calls its *MessageLoop* method. Throughout the operation of your program, the message loop processes incoming Windows messages. Your ObjectWindows programs will inherit a *MessageLoop* method that works automatically. Refine the message loop only for special dialog, accelerator or MDI handling.

The *MessageLoop* method calls three translate methods to process Windows messages. *ProcessDlgMsg* handles modeless dialogs, *ProcessAccels* handles accelerators, and *ProcessMDIAccels* handles accelerators for MDI applications. For applications that do not use accelerators or modeless dialog boxes, or which are not MDI applications, you may want to streamline the *MessageLoop*

somewhat. See the entries for the translate methods under *TApplication* in Chapter 5 of the *Windows Reference Guide.*

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 may happen as a result of the user attempting to close the main window of the application. We say it *may* happen because ObjectWindows provides a mechanism to refine an application's closing behavior; for example, to check for unsaved files before closing.

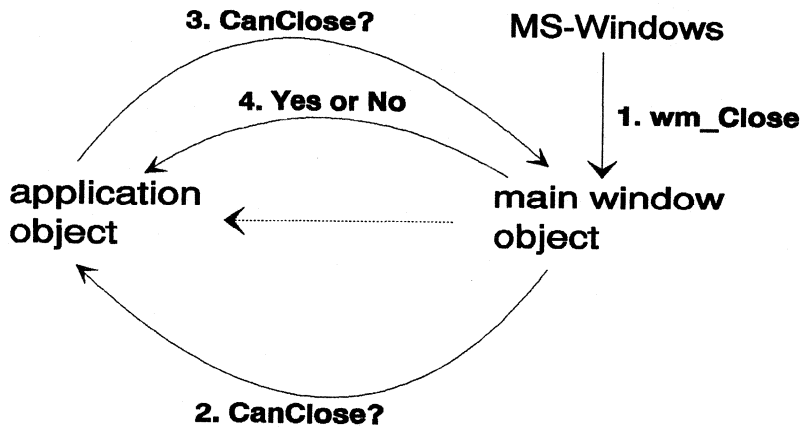
Here's what happens when the user tries to close an application by double-clicking on its Control-menu box:

1. Windows sends a *wm_Close* message to the application's main window.
2. The main window object responds by calling the *CanClose* method of the application object.
3. The application object's *CanClose* method calls the *CanClose* method of its main window.
4. The main window decides if it's okay to close itself based on whether all of its child windows say it's okay to close. If all the child windows' *CanClose* methods return *True*, the main window's *CanClose* returns *True*, and the application closes.

All window objects inherit a *CanClose* method that returns *True* by default, signifying that it's okay to close. That's why the previous *TestApp* closes without hesitation. You are free to redefine the *CanClose* methods of the application or main window type, resulting in some other closing behavior. Normally, you will refine the closing behavior of the main window object type; for example, to check to see if all files are saved.

This mechanism is comparable to announcing, "If there's anyone who knows why this application should not close, please speak now or forever hold your peace." In the end, the application object has the responsibility for giving the approval for closing the application. By default, it checks with the main window before giving this approval. This volley of method calls is depicted in Figure 8.4.

Figure 8.4
Method calls to close an application



The *CanClose* methods are Boolean functions that expect a *True* return value if the application can close. Usually a refined *CanClose* method performs some cleanup operations and, if successful, returns a *True* value.

The application type of an ObjectWindows program inherits *TApplication*'s *CanClose* method, which simply calls the *CanClose* method of its main window object. 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 user interface, or 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.

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

The TWindowsObject type

TWindowsObject is an abstract object type that unifies descendant Windows-interface object types: *TDialog*, *TWindow*, and its descendant, *TControl*. *TWindowsObject* defines behavior common to window, dialog, and control objects. *TWindowsObject* methods:

- Maintain the dual ObjectWindows/Windows object structure, including creating and destroying objects.
- Automatically support parent-child relationships between interface objects (not object inheritance relationships).
- Register new Windows classes; see Chapter 10, "Window objects."

The work of *TWindowsObject* is behind the scenes. You will rarely, if ever, derive new types directly from *TWindowsObject*, but it

serves as the foundation for the object-oriented ObjectWindows/Windows model. It defines much of the functionality an object inherits when you derive new types from *TWindow* and *TDialog*. This chapter includes an example application that registers a new Windows class, something you might have to do when designing new types of windows and dialogs.

Why interface objects?

Why do you need interface objects if Windows already has visual windows, dialogs and controls?

Each interface object normally has an associated *visual interface element*—not an object, but a physical window, dialog, or control—for which it acts as an object-oriented surrogate. The interface object provides methods for creating, initializing, managing and destroying its associated interface element, and has fields that hold data, including its interface element's handle and its parent and child windows. The interface object's methods handle 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. With ObjectWindows, you define a variable (an object) to represent a physical window, control, or dialog box that is actually managed by the Windows window manager.

The structure of an interface object, with an associated interface element, requires some awareness on your part to properly manipulate windows, dialogs and controls. For example, to create a complete interface object, you have to call *two* methods. The first is a constructor, such as *Init*, which constructs the interface object instance and sets its attributes, such as its style and menu, if any.

Window handles are explained in Chapter 7, "Object Windows overview."

The second method is the creation method, *MakeWindow*, which associates the interface object with a new interface element. This association is maintained by the interface object's *HWindow* field, which holds a handle to a window. *MakeWindow* is a method of the application object. *MakeWindow* calls the object's *Create* method, which actually tells Windows to create a visual element. *Create* also calls *SetupWindow*, which initializes the interface object, for example, to create child windows.

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

Similarly, to eliminate an interface object from your program, you must *destroy* the visual interface element and *dispose* of the corresponding interface object. The *Destroy* method, which is invoked directly by the program or indirectly when the user closes the window or its parent window (discussed later), eliminates the interface element. You can destroy the interface element without disposing of the corresponding object, for example, if you wish to create and display it again. When the interface element is destroyed, its object's handle is set to zero. You can tell if an interface object is still associated with an interface element by examining its handle.

You should note that creating both an interface object and corresponding visual element doesn't necessarily mean that you'll see something on the screen. Upon creation of the visual element, Windows checks to see if the window has the style *ws_Visible* set. *ws_Visible* and other window styles are set or cleared in the *Init* constructor, by setting or clearing them in the interface object's *Attr.Style* field. If *ws_Visible* is set, the interface element will be displayed, if *ws_Visible* is not set, the element is hidden.

At any point, an element can be shown or hidden by calling the interface object's *Show* method.

Window parent-child relationship

In a Windows application, interface elements (windows, dialog boxes, and controls) are related through parent-child links. Two interface elements are related when one is the parent window of the other, child window. Don't confuse this parental relationship with inheritance or with instance ownership, both object relationships. A child window does not inherit anything from its parent window and need not be stored in an object field of its parent window object, although it often will be.

A child window is an interface element (it doesn't have to be a *window*) that is managed by another interface element. For example, list boxes are managed by the window or dialog box in

which they appear. 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, but not controls, can be parent windows. Any interface element (dialog boxes, windows, or controls) can be a child window.

Child window lists

You must specify an interface element's parent at the time you construct it. The parent window object is a required parameter of the interface element constructor, *Init* (see Chapter 10 for an example). A child window object keeps track of its parent window object by storing a pointer to the object in its *Parent* object field. It also keeps track of *its* child window objects in a linked list, stored in its *ChildList* field. *ChildList* is maintained automatically. The child window currently pointed to by *ChildList* is the last child window created.

When you define a new interface object type with child windows, define the *Init* constructor to also construct the child window objects.

Then, when the application calls the parent window's *Create* method, the parent's interface element is created. If successful, *Create* calls the *SetupWindow* method. The *SetupWindow* inherited by all interface objects iterates over every child window in *ChildList*, calling the *Create* method for each one except dialogs, by default. Often you will redefine the *SetupWindow* of the parent window to perform some setting-up tasks after its child windows are created. Filling a list box with elements is one example. In that case, be sure the first line of your redefined *SetupWindow* method calls the inherited *SetupWindow* method.

Just as calling the parent window's *Create* method results in calling its child windows' *Create* methods, calling the parent's *Done* destructor results in calling the destructors of all its child windows. Your programs need not explicitly call *Create* or *Done* for child windows. The same is also true for the *CanClose* method, which returns *True* only after calling *CanClose* for its child windows.

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

Child window iterators

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(@CheckAllBoxes);
end;
```

Note that 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 in just the same way.

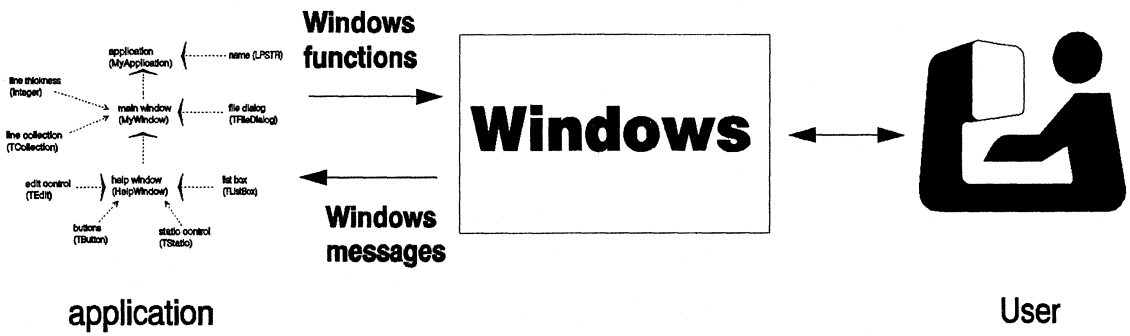
You may also want to write methods that iterate over a window's list of child windows looking for a particular child. For example, you might want to find the first check box that is checked, in a window with many check box child windows. In that case, use the *TWindowsObject.FirstThat* method as follows:

```
function TMyWindow.GetFirstChecked: PWindowsObject;

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

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

Message processing

ObjectWindows programs have a two-way communications channel with Windows. In one direction, the application controls the actions of the user interface by calling Windows functions. In the other direction, Windows sends messages back to the application in response to *events* in the program, such as the user choosing a menu selection (*wm_Command*) or clicking the left mouse button (*wm_LButtonDown*). There are over 100 standard Windows messages, and you can also create your own *user-defined* messages, as explained in Chapter 7, "Object Windows overview." Windows functions and messages constitute the two-way communication depicted in Figure 9.1.

Figure 9.1: Communications between Windows and applications



A Windows application performs processing in response to messages received from Windows. Application-specific behavior is achieved through selectively responding to the Windows messages received. Windows functions may be called in the process, and could themselves generate new Windows messages.

Incoming messages are always intended for a specific window—that window the current event most directly affects. For example, when you click the left mouse button in a particular window, the *wm_LButtonDown* message generated is intended for that window. ObjectWindows maps the incoming messages to response methods which you define for your window object types.

This interplay of Windows messages and object methods forms the skeleton of a ObjectWindows application. As you can see, the main duty of a ObjectWindows application is to respond to Windows events. This is a very different orientation than the

traditional program-controlled DOS application. This is message-based, event-driven programming. You can think of it as *not speaking until you are spoken to.* It requires discipline, but it pays off in well-behaved, multitasking applications.

Responding to messages

In order to design your window objects to respond to incoming Windows messages, you must write methods that correspond, on a one-to-one basis, to each incoming message you care to respond to. These methods are called *message-response methods.* This special message-to-method correspondence is facilitated by an extension to the virtual method definition. To mark a method (always a procedure) as a message response method, add the sum of *wm_First* and the message name (called the dynamic method index) to the end of the virtual method definition header:

```
type
  TMyWindow = object(TWindow)
    ...
    procedure WMRButtonDown(var Msg: TMessage); virtual wm_First +
      wm_RButtonDown;
    ...
```

In this example, the *WMRButtonDown* method is invoked when a *TMyWindow* object gets a *wm_RButtonDown* message. The method and message names need not be similar, but it can improve the readability of your programs. ObjectWindows uses this convention in its interface object types. *Msg* is a required variable argument typed as a *TMessage* record, which defines fields to store the *Word* (*wParam*) and *Longint* (*lParam*) parameters from the Windows message, as well as a field to store the result value (*Result*).

wParam and *lParam* carry information about the action that prompted the message. For example, a *wm_LButtonDown* message passes the x- and y-coordinates clicked on by the user in *lParam.* Often, each parameter holds two pieces of information. For example, when *wm_LButtonDown* is sent, the low-order *Word* of *lParam* holds the x-coordinate of the clicked point, and the high-order *Word* holds the y-coordinate.

For convenience, *TMessage* includes variant fields to hold the low- and high-order parts of *wParam, lParam,* and *Result. wParamLo* and *wParamHi* are *Byte* fields that hold the low- and high-order bytes of the *Word, wParam. lParamLo* and *lParamHi* are *Word* fields that

hold the low- and high-order words of the *Longint, lParam*. And finally, *ResultLo* and *ResultHi* are *Word* fields that hold the low- and high-order words of the *Longint, Result*. Note that the *Lo* and *Hi* fields are not *copies* of the original data: If you change *lParamLo*, you have changed *lParam*.

Most often, you will use the *wParam, lParamLo*, and *lParamHi* fields, all *Word* types. *Msg* is passed as a variable argument because a few Windows messages require a response that your methods will store in the *Result* field of a *TMessage* record.

In your ObjectWindows programs, you can safely ignore Windows messages to which you do not want your objects to respond, by not defining methods that respond to those messages. Those messages will be handled by default by the *DefWndProc* method your window inherits from *TWindowsObject*.

There are shortcuts to the ObjectWindows message response model for a few, very common messages associated with menus and child windows, such as edit controls and scroll bars, as outlined in the following sections.

Command and child window messages

Windows generates a *command-based message* when the user selects a menu choice or types an accelerator. It generates a *child-ID-based message* when the user activates a control, such as a button or a list box. Most of these actions result in a *wm_Command* message.

To help sort out all the possibilities and avoid a lengthy case statement, ObjectWindows automatically responds to *wm_Command* messages by generating another, more specialized, virtual method call based on the contents of the message. You define command-based and child-ID-based message response methods using a method definition extension, similar to the one already described.

Command message processing

To identify a menu or accelerator command message response method definition, use an identifier which is the sum of two constants. The first constant is the ObjectWindows-defined constant, *cm_First*. The second is the menu or accelerator ID defined in the menu or accelerator resource, and passed in the *wm_Command* message. For example, take a window, a *TFileWindow* object, that has three menu options, File | New, File | Open and File | Save. The menu IDs defined in the resource file

are 101, 102 and 103, respectively. Here's how the *TFileWindow* object definition might look:

```
TFileWindow = object(TWindow)
  ...
  procedure CMFileNew(var Msg: TMessage); virtual cm_First + 101;
  procedure CMFileOpen(var Msg: TMessage); virtual cm_First + 102;
  procedure CMFileSave(var Msg: TMessage); virtual cm_First + 103;
end;
```

To make the code more readable, you can define constants for each of the IDs. For example, you can define *cm_FileNew* as 101. Then, the method header looks like:

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

However, a window that owns a menu doesn't always have to define a respond to a command-based message. One of its child windows, including controls, can define a response instead. In fact, the child window that has the user's input focus at the time of the menu or accelerator selection gets the first opportunity to respond to a command-based message by defining a response method. Then, the message is sent to that child window's parent window, and to that window's parent, and so on, until it reaches the window that owns the menu. This means that an edit control in a window can respond to editing menu selections directly. See the type *TEditWindow* as an example of this technique.

At any point, a response method can stop the chain of messages by setting the *Result* field of the *TMessage* record parameter to 1:

```
procedure TMyEdit.CMPaste(var Msg: TMessage);
begin
  DoPaste;
  Msg.Result := 1;
end;
```

Child message processing

When the user affects a control (a child window), such as clicking on a button or typing in an edit control, its parent window or dialog object receives a *child-ID-based message*. To have the parent respond to notification messages, define a method with an identifier that is the sum of two constants. The first constant is the ObjectWindows-defined constant *id_First*. The second is the child window ID defined in a dialog resource or in the construction of a control object. For example, take a window with a list box. Define

a method called *IDListBox* to respond to the child-ID-based messages generated by the list box:

```
TListBoxWindow = object(TWindow)
  ...
  procedure IDListBox(var Msg: TMessage); virtual id_First +
    id_ListBox;
end;
```

where *id_ListBox* is a constant defined to equal the control ID. Child window IDs must be positive integers smaller than 4,094.

As an alternative to having the parent window process the child-ID-based message, you can have the control respond directly; see "Control notification messages" in Chapter 12, "Control objects."

As with command-based messages, child window messages are send to a chain of interface objects, starting with the particular child window involved. (At this point, the message is a notify-based message, as you'll learn in Chapter 12.) Then the message becomes a child-ID-based message and is passed to its parent window. The child window's response method can stop the message from reaching its parent by setting the *Result* field of the *TMessage* record parameter to 1.

Default message processing

This section applies to Windows messages you intercept directly, using the *wm_First* constant (that is, Windows messages), but not to command-based, notify-based or child-ID-based messages.

A typical Windows application receives many more Windows messages than its author might want to program responses to. Just clicking on a window, for example, might generate 10 or more messages. Knowledge of what happens when an application intercepts or ignores a message will help you to correctly program your message response methods.

When Windows sends a message to an application, it needs to know if the application intercepted and processed the message. This is because Windows has standard, built-in responses for ignored messages. For example, when the user types into an edit control, Windows automatically responds by displaying the new characters and scrolling if necessary. These standard responses are invoked by the interface object's default window procedure, its *DefWndProc* method.

When you ignore a Windows message (by not defining a method to respond to it), ObjectWindows automatically invokes the interface object's default window procedure. However, even if you intercept a message by defining a message response method, you still might need to explicitly call *DefWndProc*.

For example, the following code will intercept the *wm_Char* message, which is sent to an edit control object when the user types into an edit control. Here is the *wm_Char* message response method for our object descending from *TEdit*:

```
procedure TMyEdit.wmChar(var Msg: TMessage);
var
  TheText: array[0..255] of Char;
begin
  MessageBeep(0);
end;
```

This method responds to the user's typing action by sounding a beep. However, defining a response prohibits the standard response defined by the default window procedure. Thus, you hear the beep, but the edit control is unaffected by the user input.

What we probably wanted to do, however, was to sound the beep *and* elicit the standard response. To do this, you can explicitly call the *DefWndProc* method:

```
procedure TMyEdit.wmChar(var Msg: TMessage);
var
  TheText: array[0..255] of Char;
begin
  DefWndProc(Msg);
  MessageBeep(0);
end;
```

In the above method, the character is added and then the beep is sounded.

In general, call *DefWndProc*:

- When you want to intercept a message for notification purposes only, but have no intention of acting on the message.
- When you intercept a message to perform some action other than standard response, but you also want the standard response.
- When intercepting a message seems to disable the use of the associated interface element.

C H A P T E R

10

Window objects

This chapter explains how to create, display, and fill an application's windows. The ObjectWindows object type *TWindow* serves as a template defining most of the fundamental behavior of the program's main window and its other popup windows, if any. While instances of *TWindow* are generic windows, the type defines the protocols for opening and closing windows, as for child window and command message processing.

TWindow has three descendant types: *TMDIWindow*, *TControl*, and *TEditWindow*. 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 13. *TControl* defines controls, such as buttons and list boxes, and is covered in Chapter 12. Most often, you will 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.

The TWindow type

At a minimum, an ObjectWindows application must have a main window that it displays on the screen when the application starts up. 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 object

instance of type *TWindow* or, more often, of 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 object instances of type *TWindow* or (usually) a descendant of *TWindow*. 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*. In a well-designed application, *ZoomWindow* would probably be a specialized version of *TPaintWindow* and implemented as a descendant of *TPaintWindow* (and *TWindow*, indirectly).

Initializing and creating window objects

Like dialog and control objects, window objects are interface objects which are associated with visual interface elements. More accurately, they 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.

Initializing window objects

A typical Windows application has many different types of windows: overlapped or popup, 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 ObjectWindows-defined record type, *TWindowAttr*. Every window object stores a *TWindowAttr* record in its *Attr* field. A *TWindowAttr* includes the following fields:

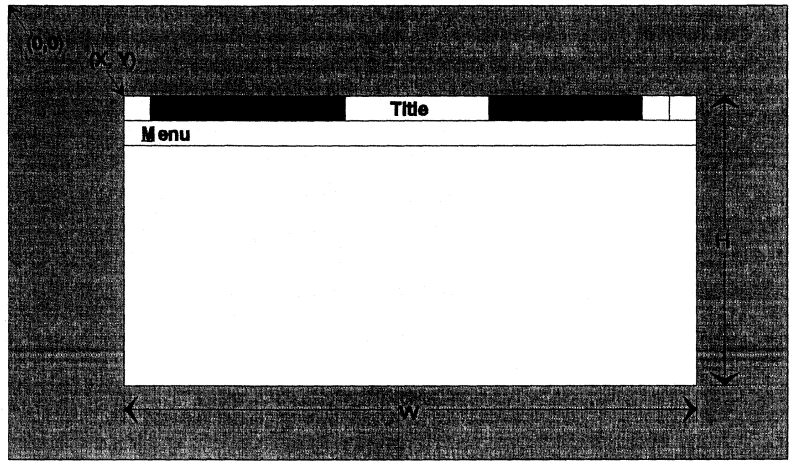
Table 10.1
Window attribute fields

Field	Usage
Title	A *PChar* that holds the title, or caption, string.
Style	A Longint that holds the combined style constant.
Menu	A handle to a menu resource (*HMenu*).
X	An Integer representing the window's initial horizontal location. It is the horizontal screen coordinate of the window's upper-left corner.

Table 10.1: Window attribute fields (continued)

Y	An *Integer* representing the window's initial vertical location. It is the vertical screen coordinate of the window's upper-left corner.
W	An *Integer* representing the window's initial width, in screen coordinates.
H	An *Integer* representing the window's initial height, in screen coordinates.

Figure 10.1
A window's attributes



Window initialization is the process of creating the ObjectWindows 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. As a default, it also sets *Style* to *ws_OverlappedWindow* **or** *ws_Visible* if the window is the application's main window (just *ws_Visible* otherwise), and *X*, *Y*, *W*, and *H* to *cw_UseDefault*, which result in a reasonably-sized overlapped window. This is the standard way to size a new main window. 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.

When you derive new window types from *TWindow*, you may define a new *Init* constructor, especially if you desire a non-default creation attribute. If you do not wish 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 AWindowType.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(100));
  AChildWindow := New(PChildWindowType, Init(@Self, 'Child Title'));
  List1 := New(PListBox, Init(@Self, id_ListBox, 201, 20, 20, 180,
    80));
  ...
end;
```

Notice, in the previous example, that the sample window's *Init* constructor is responsible for constructing its child window objects, such as popup windows and list boxes. In turn, a child window type may reset its attributes in its own *Init* constructor:

```
constructor ChildWindowType.Init(AParent: PWindowsObject; ATitle:
  PChar);
begin
  TWindow.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;
```

The *Style* attribute can be one or a combination of window style constants, such as

```
ws_OverlappedWindow
ws_PopupWindow
ws_ChildWindow
ws_Caption
ws_Border
ws_VScroll
```

The preceding list includes some popular style constants. For a complete list, see Chapter 1, "Windows styles and constants" in the *Windows Reference Guide*.

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 AWindowType.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(100));

  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

Creating window elements is the process of building the visual elements associated with a window object. This is accomplished by calling the application object's *MakeWindow* method, passing the window object as a parameter:

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

MakeWindow calls two important methods. The first is *ValidWindow*, which checks whether the window object was constructed successfully. 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 visual element. Again, if *Create* fails, *MakeWindow* returns **nil**. Otherwise, it returns a pointer to the window object.

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*. 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, "on the fly," with *MakeWindow*.

Initialization and creation summary

When designing a new window object type, you can redefine *Init* to set the window's attributes and construct child window objects, if any. (A window's attributes can also be set directly; for example, in the parent window's *Init*). *MakeWindow* builds a window object's visual element by calling *Create*, which also calls the window object's *SetupWindow* method, which automatically creates the window's child windows.

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.

Window class registration

You learned that during window object initialization, you can set 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 we'll call these *creation attributes*. There are, however, some other attributes, including background color, representative icon, and mouse cursor. To see an example of this, run Windows with a few different types of applications at once. As you move the mouse cursor from one application to another, the cursor changes from an arrow (regular) to a fat cross (for spreadsheets) to a thin cross (for graphics programs).

These attributes are more inherent to the window object type and cannot be changed during the operation of the program, as can the creation attributes. To distinguish these inherent attributes, we will call them *registration attributes*.

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 wish to change any of the window's default characteristics, you need not worry about window class registration.

If you do wish to change a registration attribute, the cursor or icon for example, you need to define a new window class by writing two methods: *GetClassName* and *GetWindowClass*. *GetClassName* is a function that simply returns the name (*PChar*) of the window class. *TWindow* defines a *GetClassName* method that 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:

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

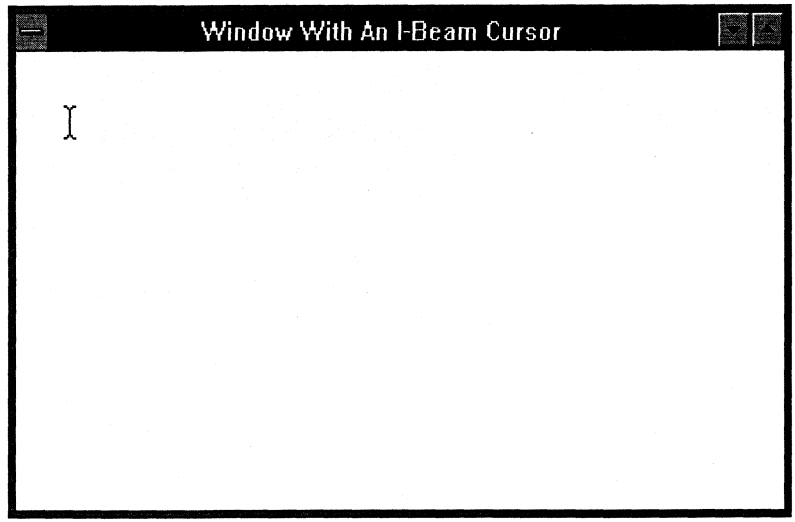
The class name may or may not be the same as the object type name. It makes no difference.

The *GetWindowClass* method takes a *TWndClass* record as a variable argument and fills its fields with new registration attributes. It is similar to defining a new *Init* method for setting a window's attributes in that you should first call *TWindow.GetWindowClass*. Then set the fields you wish to change. For example, the *hCursor* field stores a handle to a cursor resource. For *IBeamWindow*, define the following *GetWindowClass* method:

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

idc_Beam is a constant representing one of Windows' stock cursors. Figure 10.2 shows an application that uses the *IBeamWindow* object.

Figure 10.2
A program that uses the *IBeamWindow* class



Besides windows, dialog windows (not dialog boxes) need registered window classes. (See Chapter 5.) Dialog boxes and controls do not need window classes.

Registration attributes

Type *TWindow* defines a window class, 'TurboWindow', with a blank icon, an arrow cursor, and standard window color. To deviate from these characteristics, you must fill the fields of a *TWndClass* record with different data in a *GetWindowClass* method.

Here is a table showing the characteristics fields of a *TWndClass* record and their default values, as defined by *TWindow.GetWindowClass*:

Table 10.2
Window registration attributes

Characteristic	Field	Default
class style	style	*cs_HRedraw* **or** *cs_VRedraw*
icon	hIcon	LoadIcon(0, idi_Application)
cursor	hCursor	LoadCursor(0, idc_Arrow)
background color	hbrBackground	HBrush(Color_Window +1)
default menu	lpszMenuName	**nil**

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 example, *cs_HRedraw* makes the entire window redraw when its horizontal size changes; *cs_NoClose* inhibits the Close option on the Control menu; and *cs_ParentDC* gives the window its parent's display context. For a complete list of *cs_* constants, see Chapter 1, "Windows styles and constants," in the *Windows Reference Guide.*

TurboWindow class defines a window class with the style:

```
AWndClass.style := cs_HRedraw or cs_VRedraw;
```

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. As a default, *TurboWindow* class uses the stock icon, *idi_Application*, which appears as a blank rectangle.

Cursor field

The *hIcon* field holds a handle to a cursor that is used to represent the mouse pointer when it is positioned over the window. *TurboWindow* class uses the standard Windows arrow, *idc_Arrow*. Some other stock cursors include the following:

```
idc_IBeam
idc_Wait
idc_Cross
```

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 101, you would set this field with

```
AWndClass.lpszMenuName := LoadMenu(HInstance, PChar(101));
```

Edit windows and file windows

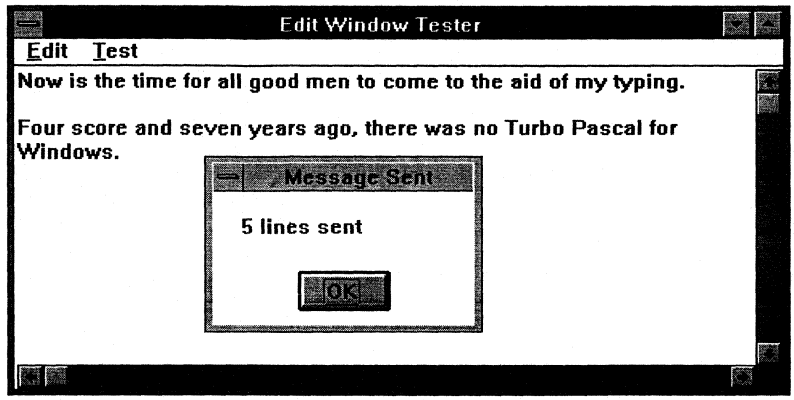
ObjectWindows provides two descendants of *TWindow* that are specialized windows for text editing. The *TEditWindow* object type provides a simple text editor that cannot read from or write to a file. The *TFileWindow* type, which descends from *TEditWindow*, provides a text editor that allows reading from and writing to files. These object types may be used directly as standard components in your applications. You can also descend your own types from them to create specialized editors. Programs using edit windows or file windows must include the *StdWnds* unit in their uses section.

Edit windows

An edit window is a window with a full-size edit control in it. *TEditWindow*'s *Init* constructor initializes the *Editor* field of *TEditWindow* to point to an edit control. *TEditWindow*'s *SetupWindow* method sets the dimensions of the edit control to that of the window's client area and creates the edit control's interface 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 following program uses an edit window to let a user edit text for a simple (non-functional) electronic mail application. Figure 10.3 shows this application.

Figure 10.3
An edit window




```
program EditWindowTester;

{$R EWNDTEST.RES}

uses WObjects, WinTypes, WinProcs, Strings, StdWnds;

const
  cm_SendText = 399;

type

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

PMyEditWindow = ^MyEditWindow;

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

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

procedure MyEditWindow.HandleSend(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;

{ Init the main window of the TestApplication - an EditWindow }

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.
```

File windows

A file window is an edit window with additional capabilities allowing it to read from and write to files. *TFileWindow's Init* constructor takes the window title as an argument. It initializes the object field *FileDialog* to point to a file dialog object. *TFileWindow* has four methods to deal with manipulating files. *Open, Save* and *SaveAs* use the *TFileWindow.FileDialog* file dialog field (see Chapter 5) 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.

Scrolling windows

Sometimes, there's more to the world than what you can see through a window. If that window is a Microsoft Windows window, you're in luck, because you can scroll that window up, down, right, or left to bring more of the window's graphics and text into view. 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, created for windows whose style constants include *ws_VScroll* (vertical scroll bar) or *ws_HScroll* (horizontal scroll bar) or both. The mechanism behind window scrolling is an object called a scroller.

TScroller is the ObjectWindows object that gives life to window scrollers, providing an automated way to scroll the text and graphics you have put into your windows. In addition, *TScroller* also scrolls your windows when the user drags the mouse outside of a window's client area; we'll call this auto-scrolling, and it works for windows that don't even have scroll bars.

Scroller attributes

A *TScroller* holds values that determine how much of a window is to be scrolled. These values are stored in the *TScroller* object 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.

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) that can 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.

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 on 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 scrolled, usually based on the size of the window's universe, such as the document being edited.

As an example, let's look at 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 chars 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.

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's constructor and store the object in the window object field *Scroller*. While doing this, you can set the size of the units and the range if known. While window scroll bars are not required for use of a scroller, as in auto-scrolling, many scrollable windows have them. To add them to a window, simply add to the window's *Attr.Style* field in its constructor to *ws_VScroll, ws_HScroll,* or both. Here is a constructor for the text editing window example:

```
constructor TScrollWindow.Init(AParent: PWindowsObject; ATitle:
PChar);
begin
```

```
    TWindow.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 will rely on their 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

Let's illustrate how to build a complete application that draws a graphics design that scrolls. This example 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. The scroll bars can be used to 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.

Here is the source:

This is SCROLAPP.PAS.

```
program Scroll;

uses Strings, WinTypes, WinProcs, WObjects;

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;

begin
  MainWindow := New(PScrollWindow, Init('Boxes'));
```

```
end;

constructor TScrollWindow.Init(ATitle: PChar);
begin
  TWindow.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.
```

Auto-scrolling and tracking

TScroller's auto-scrolling feature is enabled by default, but can be turned off by setting the *AutoMode* field in *TScroller* to *False*. 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 this is done, scrolling can only take place using the scroll bars.

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.

In addition to auto-scrolling, the example program above will "track" 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 scrolling units and range

In the examples so far, we assumed 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 **nil** values in the *TScroller* constructor.

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 example, 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);
```

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
  TWindow.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 according 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 object fields *XPage* and *YPage* 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 must 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 ScrollWindow.Init(AParent: PWindowsObject; ATitle:
PChar);
```

```
begin
  TWindow.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 Paint methods for scrolling

The example application above draws 50 rectangles but does not attempt to determine if all of the rectangles are actually visible in the window's client area. This might result in wasted effort spent drawing non-visible 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 or not 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 objects

Dialog boxes, or dialogs, are child windows that appear, usually, to perform a specific task, such as configuring a printer or accepting a text entry. The *TDialog* object type supports the initialization, creation, and execution of all types of dialog boxes. As with *TWindow*, you can define dialog types that descend from *TDialog* for each type of dialog in your application. ObjectWindows already supplies object types for the two most common dialogs, text input, and file dialogs. ObjectWindows also supplies a type, *TDlgWindow*, that allows you to create a dialog that behaves more like a window.

Using dialog resources

Like menus, dialogs are designed and specified using a resource description created outside of the program code. The dialog 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 and does not describe its behavior. That is the application's responsibility. Associated with a dialog resource is an identifier, which can be an ID number (*Word*) or a string (*PChar*). Inside an application, a dialog object is matched up to a dialog resource through the resource ID.

Using dialog objects

Using dialog objects is a lot like using popup window objects. Dialogs are child windows of a parent window. For simple dialogs that appear for only a short time, all of the dialog handling can be done by one method of the parent window object. The dialog can be constructed, executed and disposed all in one method, with no need to store the dialog in an object field. For more complex dialogs, you might want to store the dialog in an object field of your dialog type.

Like popup windows and controls, dialogs are child windows, and are added to the *ChildList* of their parent windows when constructed.

Constructing and initializing dialogs

Dialog objects are constructed and initialized with an *Init* constructor. *Init* takes a pointer to the parent window and a *PChar* representing the dialog resource name as its parameters:

```
ADlg := New(PSampleDialog, Init(@Self, 'EMPLOYEEINFO'));
```

or:

```
Dlg := New(PSampleDialog, Init(@Self, PChar(120)));
```

Executing dialogs

Executing, or running, a dialog is analogous to creating and displaying a window. However, since dialogs are usually displayed for a shorter time period, some of the steps can be abbreviated. This is dependent on whether the dialog will be displayed as a modal or modeless dialog.

Modal and modeless dialogs

Modal dialog boxes are the most common dialog boxes. Similar to the message boxes generated from the *MessageBox* function, modal dialogs are displayed for a specific purpose for a short period of time. *Modal* means that while the dialog is displayed, the user cannot select or use its parent window. The user must use the dialog and click on the OK or Cancel buttons to destroy the dialog and proceed with the program. A modal dialog, in effect, freezes the operation of the rest of the program. Thus, with one method call, *ExecDialog*, you can create, display and end a dialog:

```
ReturnValue := Application^.ExecDialog(ADlg);
if ReturnValue = id_OK then
  { Code to retrieve data and process dialog }
else
  if ReturnValue = id_Cancel then { Cancel was pressed }
```

If the user clicks the dialog box's Cancel button, *ExecDialog* returns *id_Cancel*.

A modeless dialog 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 at any time, as long as the dialog object still exists. This is most often done in the *OK* method, which is the method called when the user presses the OK button.

Ending dialogs

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.

Managing dialog objects

Dialogs differ from other child windows, such as popup windows and controls, in that they are often displayed and destroyed many

times during the life of their parent windows. Dialogs 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 some other event.

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.

In the case of modal dialogs, it is probably best to construct, execute, and dispose of the objects all in one method of the parent window object, as the examples in this chapter demonstrate. That way, every time the dialog appears, it is a new object.

The case of modeless dialogs is more like that of popup windows and controls. The main reason that you cannot dispose of modeless dialogs right after running them is that, unlike their modal counterparts, you can't know when the user will close the dialog box. (Remember that with modal dialogs, the *ExecDialog* method does not return until the dialog is closed.) Thus, it is best to construct a modeless dialog in the constructor of its parent window and store it in an object field of the parent. Unlike window and control objects used as child windows, dialogs are not automatically displayed when their parent windows are displayed.

```
constructor ParentWindow.Init(AParent: PWindowsObject; ATitle:
PChar);
begin
  TWindow.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;
```

Finally, the single dialog object will be automatically disposed with its parent window is disposed.

Control manipulation and message processing

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. To see how to use control objects in a window (not a dialog box), see Chapter 12. 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 (a dialogue, you might say) 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.

Manipulating dialog controls

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, get it with the *GetItemHandle* method:

```
GetItemHandle(id_LB1);
```

Responding to control notification messages

In the other direction, when the user clicks on a control, such as pressing a button or making a selection in a list box, a special child-window-based message called a *control notification message* is received by the control's dialog. Define a child-ID-based message response method in the parent dialog type for each child control:

```
TestDialog = object(TDialog)
  procedure HandleBN1Msg(var Msg: TMessage); virtual id_First +
    id_BN1;
  procedure HandleListBox(var Msg: TMessage); virtual id_First +
    id_LB1;
end;
```

Each control notification message comes with a notification code, an integer constant, which identifies the action which 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 on 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 in *Msg.lParamHi*. To intercept a control notification message, write the response method for the dialog type to handle the important notification codes:

```
procedure TestDialog.HandleLB1Msg(var Msg: TMessage);
begin
  case Msg.lParamHi of
    lbn_SelChange: {Handle selection change};
    lbn_DblClk: {Handle selection double-click};
  end;
end;
```

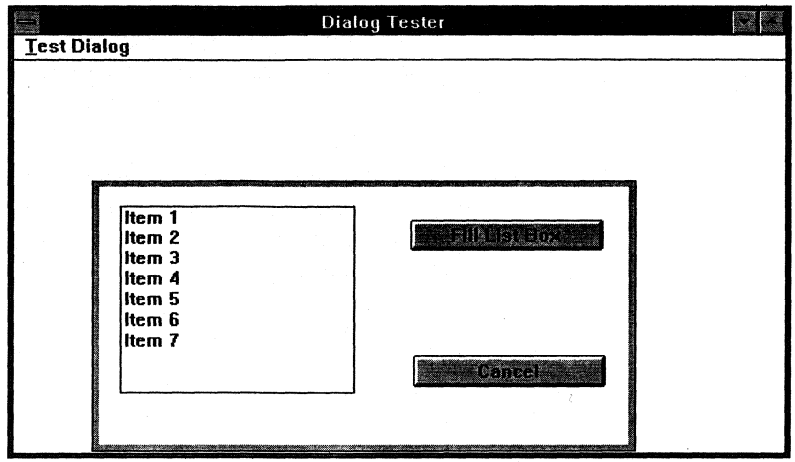
Example of dialog/control communication

Listed here is a program whose main window brings up a modal dialog, defined by the dialog type, *TestDialog*. This program features a two-way communication between the dialog object and its controls. *TestDialog*'s two methods, *HandleBN1Msg* and *HandleLB1Msg* are child-ID-based response methods which are invoked when controls (child windows) are clicked on by the user. For example, when the user clicks on the dialog's *BN1* ('Fill List Box') button, the method *HandleBN1Msg* is invoked. Similarly, when the user clicks on the list box, *HandleLB1Msg* is invoked. In the other direction, the code of the *HandleBN1Msg* method sends the dialog a control message, *lb_AddString*, using

the dialog method *SendDlgItemMsg*, in order to fill the list box with text items.

This program also shows how to create a new dialog, by defining a new dialog type and associating it with a dialog resource in the call to the constructor *Init* in the *TestWindow.RunDialog* method. Figure 11.1 shows the program.

Figure 11.1
A dialog application



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 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.

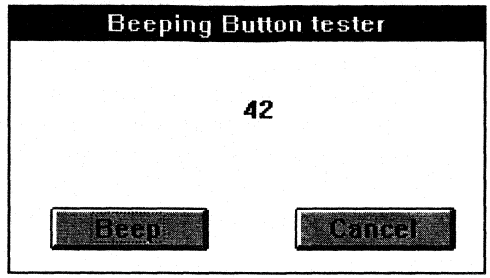
Note also that, unlike setting up a window object, which takes two steps (*Init* and *MakeWindow*), associating an object with a control element is a single-step process, because the control element already exists: It's loaded from the dialog resource. You just have to tell *InitResource which* control from the resource you want to associate the object with, using the control's ID.

Calling DefWndProc

In each of the control object's methods the last statement generally should be a call to *DefWndProc(Msg)* so that the standard Windows message handling can take place. If you choose not to call *DefWndProc*, then your method will be the *only* action taken for the particular message. Your method will be responsible for handling (or ignoring) such taken-for-granted tasks as showing a button depression or passing a message to a control's parent. If you call *DefWndProc*, your method enhances the normal processing; if you don't call *DefWndProc*, your method replaces the normal processing. One case in which you might want to override default processing for a message is if it returns a particular value and you want your method to return a different value.

Here is a sample application which brings up a dialog with a button that has an associated object to modify its message responses. The button itself generates a beep when it is pressed and then calls *DefWndProc*. This allows the dialog to also get the message and display the number of times the button was pressed. Figure 11.2 shows the program.

Figure 11.2
A dialog with a specialized button



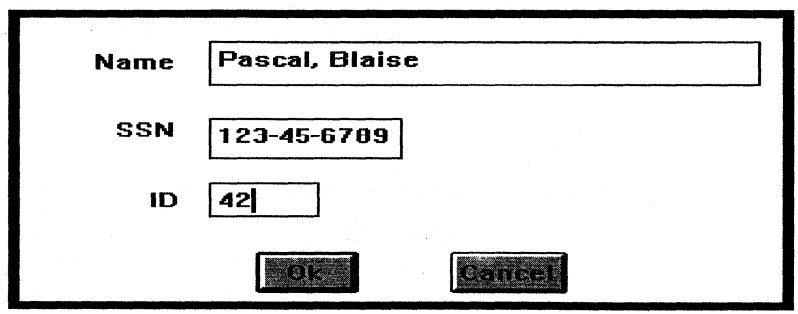
The complete program, INIRTEST.PAS, can be found on your distribution disks.

Extended example of using dialogs

Listed here is an extended example that shows how to define a complex dialog that solicits the user for text input and then validates the input. It displays a dialog box with edit fields for an employee's name, social security number, and employee ID. When the user clicks OK, it verifies that the name includes no digits, that the social security number is the correct length, and that the employee ID is an integer.

The new dialog type, *EmployeeDlg*, redefines the *CanClose* method to validate the input. Figure 11.3 shows this application.

Figure 11.3
A dialog that validates user input



The complete listing of VDLGTEST.PAS can be found on your distribution disks.

Dialog windows

After working with dialogs and windows, you might wonder, "What is the difference between them?" The major difference 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 dialogs 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 dialogs some of the more flexible features of windows.

TDlgWindow descends from *TDialog* and inherits its methods, such as *Execute*, *Create*, *Ok*, and *EndDlg*. Like dialogs, dialog windows have a corresponding dialog 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 should also be listed in the dialog resource.

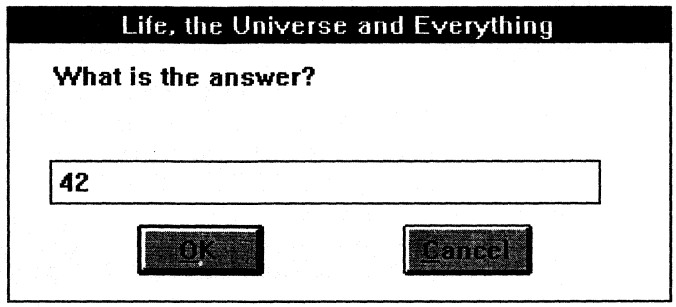
In most cases, you will run dialog windows as you would windows or modeless dialogs, using the *Create* and *Show* methods, rather than the *Execute* method.

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.

Input dialogs

Input dialogs are stock dialogs provided with ObjectWindows . They are simple dialog objects, defined by type *TInputDialog*, that prompt the user for a single line of text input (see Figure 11.4).

Figure 11.4
An input dialog



You can run input dialogs as either modal or modeless dialogs, but you'll usually run them as modal. Associated with an input dialog object is an input dialog resource, provided in ObjectWindows in the file STDDLGS.RES. Be sure this resource is included in your application. Using the *StdDlgs* unit will automatically include the resources in STDDLGS.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 which gets filled with the user's input when the user clicks on the OK button.

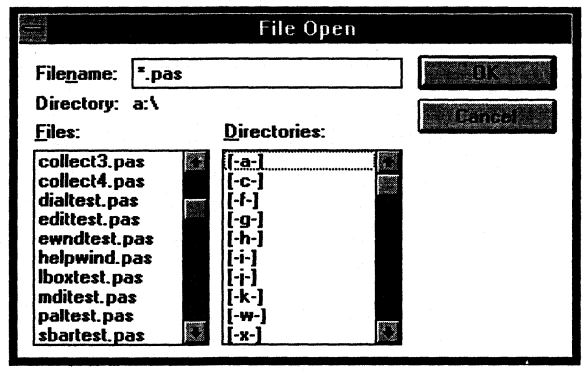
When the user clicks OK or presses *Enter*, the line of text entered in the input dialog is automatically transferred into the character array that passed in the default text. This example constructs and displays an input dialog and retrieves the text:

```
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;
```

File dialogs

File dialogs are another type of stock dialog provided with ObjectWindows in the type *TFileDialog*. Use a file dialog every time you want to prompt the user for a file name, such as in the File Open and Save As functions many applications feature. See Figure 11.5.

Figure 11.5
A file dialog



In most cases, run file dialogs as modal dialogs (see "Running file dialogs" on page 151). Associated with a file dialog object is a file dialog resource, provided in ObjectWindows in the file STDDLGS.RES. Using the unit *StdDlgs* will automatically include the resource file.

Initializing file dialogs

TFileDialog.Init allows you to specify a parent window, a resource to use (which determines whether the dialog is an Open or a Save As file dialog), and a null-terminated string which passes in the default value of the edit text in the dialog. This same string holds the selected file name if the dialog is closed. If the dialog is cancelled, the string is unchanged.

Here is a typical use of a file dialog to open a text file:

```
var
  AFile: array[0..fsPathName] of Char;
begin
  StrCopy(AFile, '*.TXT');
  if Application^.ExecDialog(New(PFileDialog, Init(@Self,
    PChar(sd_FileOpen), AFile))) = id_OK then
      ...                                    { AFile holds file name }
end;
```

Running file dialogs

There are actually two flavors of the file dialog: the open file dialog and the save file dialog. The difference is the presence (or absence) of a list box with file names.

When you run the dialog with *ExecDialog*, you will get the type of dialog specified by the resource template (the second parameter of the *Init*). If the resource contains a file list box (control ID *id_FList*), the dialog is a File Open dialog. If control ID *id_FList* does not exist, the dialog is run as a File Save As dialog. You can create your own file dialog resources. The resource file STDDLGS.RES contains two templates, one for File Open (*sd_FileOpen*), and one for File Save As (*sd_FileSave*).

Here is an example of typical file dialog use:

```
procedure TMyWindow.SaveAs(var AFile: array[0..fsPathName] of Char);
begin
  if Application^.ExecDialog(New(PFileDialog, Init(@Self,
    PChar(sd_FileSave), AFile))) = id_OK then
    begin
      { AFile is now a full path name }
      SaveFile(AFile);                    { write the file to disk }
    end;
end;
```

C H A P T E R

12

Control objects

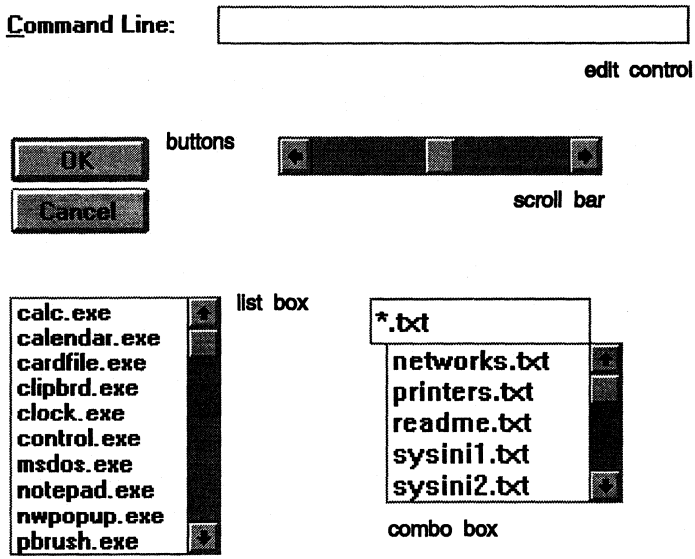
Most Window applications make use of the user interface devices collectively called controls. The following table describes the Windows controls supported by ObjectWindows object types:

Table 12.1
Windows controls supported by ObjectWindows

Control	Object type	Use
list box	*TListBox*	A scrollable list of items, such as files, from which to choose.
scroll bar	*TScrollBar*	A standalone scroll bar similar in appearance 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 input text.
static control	*TStatic*	A piece of displayable text that cannot be modified by the user.
combo box	*TComboBox*	A combination of a list box and edit control.
MDI client	*TMDIClient*	Described in Chapter 13.

Figure 12.1 shows the screen appearance of some typical controls.

Figure 12.1
Some sample controls



ObjectWindows defines object types for all of the control types listed above, to be used as child windows inside other windows. Controls that are a part of a dialog box are control interface elements specified in a dialog resource and need no corresponding ObjectWindows object (see Chapter 11). This chapter outlines the functionality of control objects and shows how they might be used inside of a window.

The TControl object type

To Windows, controls are specialized windows. In ObjectWindows, type *TControl* descends from type *TWindow*. 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 windows in the way they respond to Windows (*wm_*) messages.

Also, you may find that the ObjectWindows control types supply all of the functionality required by your application, so most often you will use the existing types unchanged. You can use the existing controls in an unlimited number of combinations, based on the functionality of your program. There are some cases where

you might want to define descendant control types. For example, you can define a specialized list box type, called *StatesListBox*, that stores the names of the 50 states and automatically displays them whenever created.

Type *TControl*, like *TWindowsObject*, is an abstract object type. You will instantiate its descendants, *TListBox*, *TButton*, and the others. This section describes the general behaviors of all control objects.

Constructing and creating control objects

The life of a control usually parallels the life of its parent window. In a typical scenario, the parent window object defines a data field for each of its child windows. If you created a window type called *SampleWindow*, it might have a data field, called *LB1*, to hold a *TListBox* object.

In addition to data fields, parent windows also maintain a list of their child windows in the *ChildList* field. All of a window object's child controls are automatically added to this list, so they do not require individual data fields. However, it may be more convenient to access and manipulate control objects when they have corresponding object fields. Controls that are rarely manipulated, such as static controls or group boxes, should not have fields dedicated to them. With other controls, you can decide whether it is easier to handle the particular control with a pointer in its parent window.

There are two steps to creating controls. The first is to construct the control object and the second is to create the corresponding control element. Both steps are usually implemented within the construction and creation steps of the parent window object. The *Init* method of the parent window usually calls the constructors of its child controls. For example, this window type's *Init* method constructs a new list box object and stores it in an object field, *LB1*:

```
constructor SampleWindow.Init(AParent: PWindowsObject; ATitle:
PChar);
begin
  TWindow.Init(AParent, ATitle);
  LB1 := New(PListBox, Init(@Self, id_LB1, 20, 20, 100, 80));
end;
```

The six fields in the list box *Init* constructor represent the control's parent window object, the control's ID, the x- and y- screen coordinates of its upper left corner (relative to its parent window's client area), its width (*W*), and its height (*H*), respectively. All

ObjectWindows-supplied descendants of *TControl* take at least these six parameters, except for *TMDIClient* (see Chapter 13, "MDI objects").

Control objects that have no corresponding individual object field are constructed as follows:

```
constructor TSampleWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
var TempLBox: PListBox;
begin
  TWindow.Init(AParent, ATitle);
  TempLBox := New(PListBox, Init(@Self, id_LB1, 20, 20, 100, 80));
end;
```

All *TControl* descendants except *TMDIClient* get the styles *ws_Child* and *ws_Visible* from calling *TControl.Init*. If you want to change a control's style you can manipulate its *Attr.Style* field as described for windows in Chapter 11.

A control object's control element is automatically created by the *SetupWindow* method inherited by the parent window object. You need *not* explicitly call *Create* for your control objects.

The controls are also filled and set, if necessary, in *SetupWindow*. Here is a typical *SetupWindow* method implementation:

```
procedure SampleWindow.SetupWindow;
begin
  TWindow.SetupWindow;                    { creates child controls }
  { Add items to list: }
  LB1^.AddString('Item 1');
  LB1^.AddString('Item 2');
end;
```

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 display and hide controls.

To review control creation, all you are required to do is to optionally define fields in your parent window object type to hold each child control. Then, for your window type, define an *Init* constructor that creates the control object and an *SetupWindow* that sets up the control object. Notice that all the work is done by the parent, while the children go along for the ride. That sounds about right.

Destroying and disposing of controls

Like creating controls, disposing of them is the parent window's responsibility. The control element is automatically destroyed along with the parent window element when the user closes the window or application. The corresponding control object is automatically disposed by the parent window's destructor.

Controls and message processing

Communication between a window object and its control objects is similar in some ways to the communication between a dialog object and its control elements. Like a dialog, 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

Dialogs manipulate their controls by sending them messages with the *SendDlgItemMsg* method, with a control message name, like *lb_AddString*, as a parameter (see Chapter 11). However, control objects simplify this process by offering 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:

```
LB1^.AddString('Scotts Valley');
```

However, manipulating a control object without an object field requires extracting the object from the parent window's child window list. Use the *GetChildWithID* method:

```
var
  TheListBox: PListBox;
begin
  TheListBox := PListBox(ChildWithID(id_LB1));
  TheListBox^.AddString('Scotts Valley');
end;
```

Responding to control notification messages

Events in a window's controls result in *control notification messages* (see "Responding to control events" in Chapter 6). In most cases, window objects respond to control notification messages by invoking child-ID-based response methods. To do this, define child-ID-based message response methods in the parent window type for each child control:

```
TSampleWindow = object(TWindow)
  LB1: PListBox;
  BN1, BN2: PButton;
  ST1: PStatic;
  procedure IDLB1(var Msg: TMessage); virtual id_First + id_LB1;
  procedure IDBN1(var Msg: TMessage); virtual id_First + id_BN1;
  procedure IDBN2(var Msg: TMessage); virtual id_First + id_BN2;
end;
```

In your response methods, *Msg.lParamHi* carries the control notification code, such as *lbn_SelChange* and *bn_Clicked*. Write a response to handle the important notification codes:

```
procedure TSampleWindow.IDLB1(var Msg: TMessage);
begin
  case Msg.lParamHi of
    lbn_SelChange: { Handle selection change };
    lbn_DblClk: { Handle selection double-click };
  end;
end;
```

There are times, however, when you want a control object itself to respond to a control notification message, thereby building a response behavior into the control. For example, you might want to design a button that changes its text when the user clicks on it, or an edit control that allows only digits to be entered. You can do this for controls in windows because they are control objects. Control elements in dialogs cannot respond to notification messages unless they are associated with control objects (using *InitResource*).

In these special cases, you want the control type to supply the desired behavior, rather than duplicating it in each parent window. The result is a standalone object type that can be reused many times in many applications.

To program a control object to directly respond to its notification messages, define for its type a *notify-based* response method (rather than a child-ID-based response method for its parent window object). Use the sum of *nf_First* and the control notification code as a method header identifier. To do this, define a type to descend from an existing control type:

```
TSpecializedLB = object(TListBox)
  procedure LBNSelChange(var Msg: TMessage); virtual nf_First +
    lbn_SelChange;
```

```
  procedure LBNDblClk(var Msg: TMessage); virtual nf_First +
    lbn_DblClk;
end;

procedure TSpecializedLB.LBNSelChange(var Msg: TMessage);
begin
  {Handle selection change};
  Msg.Result := 1;
end;
```

If you do not wish for the parent window to receive a notification message after the control has processed it, set the result of the message (*Msg.Result*) to 1. If you do not set the result to 1, you can perform additional response processing from the parent window object.

Windows that act like dialogs

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.

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. For example, you can ask the user to choose from a list of files, printers, shapes, or fruit, depending on the nature of the program. 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 and creating list boxes

TListBox's *Init* constructor takes a 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.Init calls *TControl.Init*, which gives the control the style *ws_Child* or *ws_Visible*. It then adds the style *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 wish to use a different list box style, you can modify *TListBox's Attr.Style* field. For example, if you want 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;
```

or

```
LB1 := New(PListBox, Init(@Self, id_LB1, 20, 20, 340, 100));
LB1^.Attr.Style := ws_Child or ws_Visible or lbs_Notify or ws_VScroll
  or ws_Border;
```

As with all child windows, list box control objects are automatically created by their parent windows.

Modifying list boxes

After you create a list box, you need to fill it with list items, which must be strings. Later, you might want to add or remove items, or clear the list completely. To fill or add items, call the list box's *AddString* method:

```
LB1^.AddString('Item 1');
LB1^.AddString('Item 2');
LB1^.AddString('Item 3');
LB1^.AddString('Item 4');
LB1^.AddString('Item 5');
LB1^.AddString('Item 6');
```

If *AddString* is unsuccessful, it returns a negative value.

A list box keeps a list of strings much like an array of strings; both keep their elements at an index. If *LB1* were an array, 'Item 1' would be stored at *LB1[0]*. 'Item 2' would be stored at *LB1[1]*, 'Item 3' at *LB1[2]*, and so on. Every time you call *AddString*, the specified string is added in alphabetical order by default, or at the end of the list if the style excludes *lbs_Sort*.

Regardless of whether your list box is sorted, you can also insert a new element at a specified index with *InsertString*:

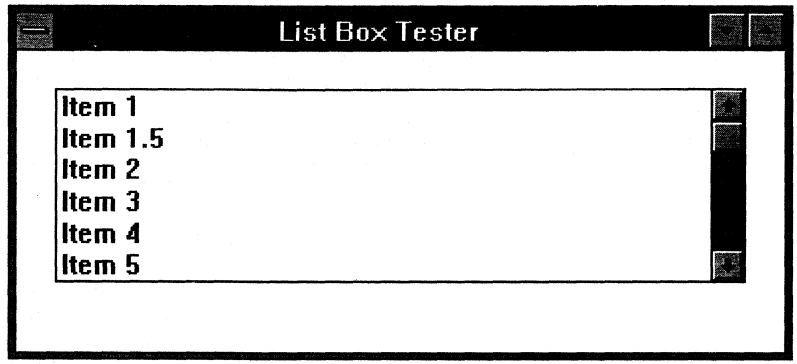
```
LB1^.InsertString('Item 1.5', 1);
```

This also moves the strings stored at each index greater or equal to 1 up one notch. The seven preceding method calls result in the following list:

Index	Item
0	'Item 1'
1	'Item 1.5'
2	'Item 2'
3	'Item 3'
4	'Item 4'
5	'Item 5'
6	'Item 6'

See Figure 12.2.

Figure 12.2
A filled list box



Inserting a new string at index –1 appends the string to the end of the list.

To remove an element, call *DeleteString*. The following method call deletes the string at index 1 ('Item 1.5') and moves the strings at an index higher than 1 down one index:

```
LB1^.DeleteString(1);
```

Finally, *ClearList* deletes every string in the list box:

```
LB1^.ClearList;
```

Querying list boxes

There are three methods you can call to find out information about the list held by a list box object. *GetCount* returns the number of items in the list. *GetString* gets the string located in the list at the index specified by the integer argument. The *PChar* argument points to a buffer to receive the string. *GetString* also

returns an integer representing the length of the string. *GetStringLen* simply returns the length of the string at the specified index, but doesn't pass the string itself.

Getting selections from a list box

There are only a few things the user can do with a list box: scroll through the list, single-click on an item, and double-click on an item. When a list box user action takes place, Windows sends a *list box notification* message to the list box's parent window. Normally, you will define a child-ID-based method in the parent window type to handle messages for each of the parent's controls.

Every list box notification (*lbn*) message contains a list box notification code (an integer constant) in *Msg.lParamHi* to specify the nature of the action. The following table summarizes the most common *lbn* codes:

Table 12.2
List box notification messages

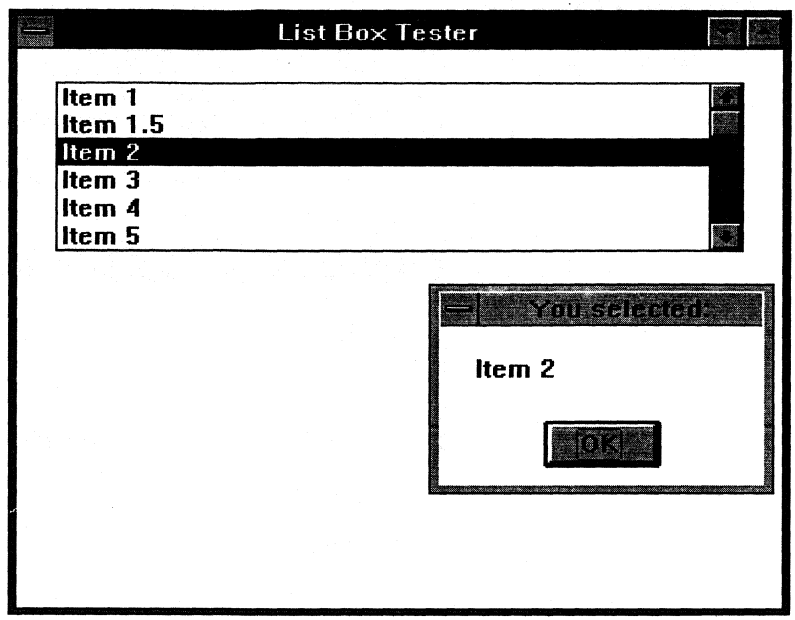
wParam	Action
lbn_SelChange	An option has been selected with a single mouse click.
lbn_DblClk	An option has been selected with a double mouse click.
lbn_SetFocus	The user has given the list box the focus by clicking or double-clicking on an item, or by using *Tab*. Precedes *lbn_SelChange*.

Here is a sample parent window method to handle the list box messages:

```
procedure TLBoxWindow.HandleLB1Msg(var Msg: TMessage);
var
  Idx: Integer;
  ItemText: string[10];
begin
  if Msg.lParamHi = lbn_SelChange then
  begin
    Idx := LB1^.GetSelIndex;
    if LB1^.GetStringLen(Idx) < 11 then
    begin
      LB1^.GetSelString(@ItemText, 10);
      MessageBox(HWindow, @ItemText, 'You selected:', mb_OK);
    end;
  end
  else DefWndProc(Msg);
end;
```

The user has made a selection if *Msg.lParamHi* equals the *lbn* constant *lbn_SelChange*. If so, it gets the length of the selected string, verifies that it will fit into a 10-character string, gets the string, and shows it in a message box (see Figure 12.3).

Figure 12.3
Responding to the user selecting a list box item



Selected items are manipulated by four *TListBox* methods: *GetSelString*, *GetSelIndex*, *SetSelString*, and *SetSelIndex*. The *Get* methods get the string and index of the selected string. The *Set* methods bypass the user and force the selection of a particular item, by specifying the string or index. This causes it to come into view, if it isn't already.

Example program: LBoxTest

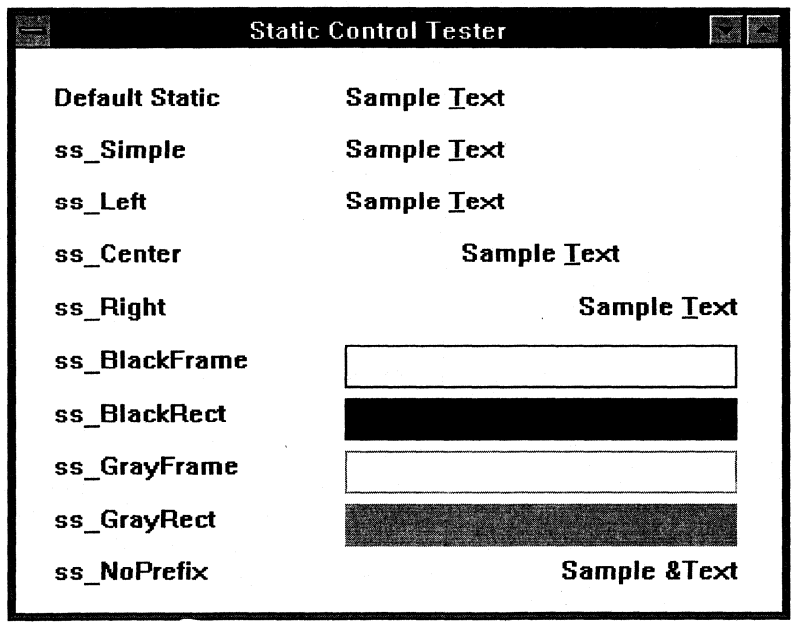
The program *LBoxTest* is a complete program that creates a window with a list box. When the application is started, the main window appears with the list box. When the user chooses a list box item, a dialog appears with the list item displayed. Notice the relationship between the window object and the list box object. The list box is not only a child window of the main window, but it is also *owned* by the main window as an object field. One of the main window's object fields is *LB1*, which holds the list box object.

The complete file, LBOXTEST.PAS, can be found on your distribution disks.

Static controls

Static controls are usually-unchanging units of text or simple graphics that can appear on the screen in a window or dialog. The user does not interact with statuc controls, although the program can change their text. Figure 12.4 shows a variety of static control styles and their corresponding Windows style constant.

Figure 12.4
Static controls



Constructing static controls

Since the user never interacts directly with a static control, the application rarely, if ever, receives control notification messages concerning a static control. Therefore, most static controls can be constructed with –1 or some other unused number as the resource ID, and they can all use the same one.

In addition to the usual *Init* parameters for a control, *TStatic.Init* takes one additional parameter, the text length, which sets the maximum length of text that can fit into the control. 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.

TStatics's constructor *Init* constructs new static control objects.

```
Stat1 := New(PStatic, Init(@Self, id_ST1, '&Text', 20, 50, 200,
  24, 6));
```

Often, you will not access or manipulate a static control object once you have created it. For this reason, it is often unnecessary to assign an object field to hold the static control object, and a temporary variable will suffice:

```
TempStat := New(PStatic, Init(@Self, id_ST1, '&Text', 20, 50, 200,
  24, 6));
```

Init produces a static control with the *ss_Left* (left justified) style, as well as *ws_Child* and *ws_Visible*. To change the style, manipulate the *Attr.Style* field:

```
Stat1^.Attr.Style := Stat1^.Attr.Style and (not ss_Left) or
  ss_Center;
```

One available option for static controls is to underscore one or more characters in the text string. The implementation and effect of this is similar to underscoring the first character of a menu choice: Insert an & character in the string immediately preceding the character to be underscored. For example, to underscore the T in 'Text', send the string '&Text' in the *Init* call. If you really want the & character in the string, use the Windows static style, *ss_NoPrefix* (see Figure 12.4).

Querying static controls

To inquire about the current text held by a static control, use the *GetText* method.

Modifying static controls

TStatic has two methods for altering the text of a static control. *SetText* sets the static's text to be the passed *PChar* argument. *Clear* erases the static's text. However, you cannot change the text of static controls created with the style *ss_Simple*.

Example: StatTest application

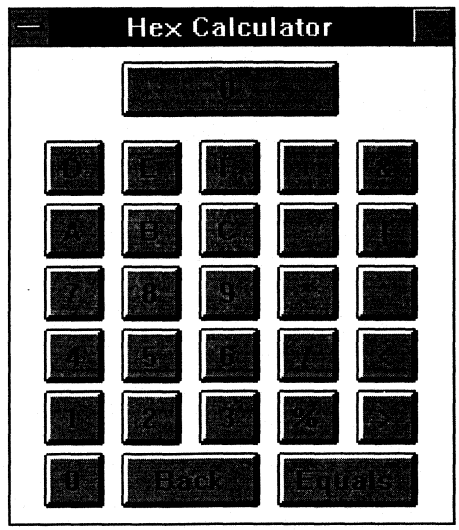
The program *StatTest* creates the static testing application shown in Figure 12.4. Note that the labels, such as 'Default Static' and 'ss_Simple', are static controls, as well as 'Sample Text' and the black and gray boxes.

The complete file, STATTEST.PAS, can be found on your distribution disks.

Push button controls

Push buttons (sometimes called "command buttons") are used to perform some task each time the button is pressed. There are two styles of push buttons, both of type *TButton*. The two styles are *bs_PushButton* and *bs_DefPushButton*. Default push buttons are similar to push buttons but have a bold border and are generally used to indicate the default user response. Figure 12.5 shows a sample Windows program that uses plain and default push buttons.

Figure 12.5
A Windows program that uses buttons



TButton's *Init* constructor takes the parent window, control ID, a text string of type *PChar*, the button's location and size, and a

Boolean flag, *IsDefaultButton*, which indicates whether the button should be a default push button or a regular push button. A default push button has a dark border, and is the one activated by default when *Enter* is pressed.

```
Push1 := New(PButton, Init(@Self, id_Push1, 'Test Button', 38, 48,
  316, 24, False));
```

Responding to button messages

Whenever the user clicks on a push button, the button's parent window receives a child-ID-based 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 with any other program-controlled activity. To respond to button messages, define a child-ID-based method to handle each button in question. For example, the following method, *IDBut1*, handles the response to the user clicking on a push button. The only notification code defined by Windows for push buttons is *bn_Clicked*, so you don't need to check the notification code.

```
TTestWindow = object(TWindow)
  But1: PButton;
  procedure IDBut1(var Msg: TMessage); virtual id_First + id_But1;
  ...
end;

procedure TestWindow.IDBut1(var Msg: TMessage);
begin
  MessageBox(HWindow, 'Clicked', 'The Button was:', mb_OK)
end;
```

Check boxes and radio buttons

Type *TCheckBox* descends from *TButton* and type *TRadioButton* descends from *TCheckBox*. Check boxes are generally used to present the user with a two-state option. The user can check or uncheck the control, or leave it as is. When there is a group of check boxes, more than one may 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, radio buttons might be used to pick a font for a particular character. We will sometimes refer to check boxes and radio buttons collectively as *selection boxes*.

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 on a selection box, it is considered an event, resulting in a Windows message. As with other controls, these messages are usually intercepted and acted upon by the selection box's parent window. However, you may wish 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

The *Init* constructors for check boxes and radio buttons take a control ID, text, location, size, and *AGroup*. *AGroup* is a pointer to a group box object (see "Group boxes" on page 169) which is used to logically group the check boxes or radio buttons. If *AGroup* is **nil**, the selection box is not part of any logical group.

```
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 are initialized, by default, to 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. For example:

```
ChBox1^.Attr.Style := ChBox1^.Attr.Style and not bs_AutoCheckBox or
bs_CheckBox;
```

Querying a selection box's state

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 of type *TCheckBox* to query the state of a selection box:

```
MyState := Check1^.GetCheck;
```

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.

Modifying a selection box's state

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. The *TCheckBox* type defines four methods for modifying a check box's state: *Check*, *Uncheck*, *Toggle*, and the most general, *SetCheck*.

Check forces the check box's state to be checked:

```
Check1^.Check;
```

Uncheck forces the check box's state to be unchecked:

```
Check1^.Uncheck;
```

Toggle changes the check box's state from checked to unchecked, or vice versa. In the case of 3-state buttons, *Toggle* changes an unchecked box to checked, a checked box to grayed, and a grayed box to unchecked.

```
CheckBox1^.Toggle;
```

SetCheck gives you complete control over the state of the check box:

```
Check1^.SetCheck(0);                    { Unchecks button }
Check1^.SetCheck(1);                      { Checks button }
```

When these methods are used by radio buttons, ObjectWindows ensures that only one radio button per group is checked, as long as the buttons are assigned to a group.

Group boxes

In its simplest form, a group box is a labeled static rectangle that visually groups other controls. A group box's *Init* constructor takes a parent window, an ID, text, a location, and a size:

```
GroupBox1 := New(PGroupBox, Init(@Self, id_GB1, 'A Group Box', 38,
  102, 176, 108));
```

While a group box visually associates a group of other controls, it can also logically associate a group of selection boxes (checkboxes 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.

Responding to group box messages

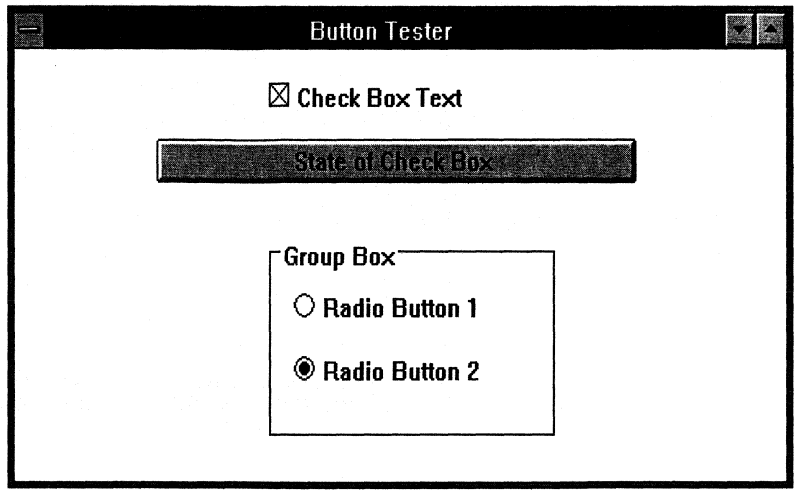
When an event occurs that may have changed the group box's selections (for example, when a user presses a button or the program calls *Check*), a child-ID-based message will be received by the group box's parent window. The parent intercepts the message by using the sum of *id_First* and the group box 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

BtnTest is a full program that creates a window with push button, check box, radio button, and group box controls. When the application is started, the main window appears with the controls. When the user clicks on the controls, the application responds in a variety of ways. See Figure 12.6.

Figure 12.6
A window with various buttons

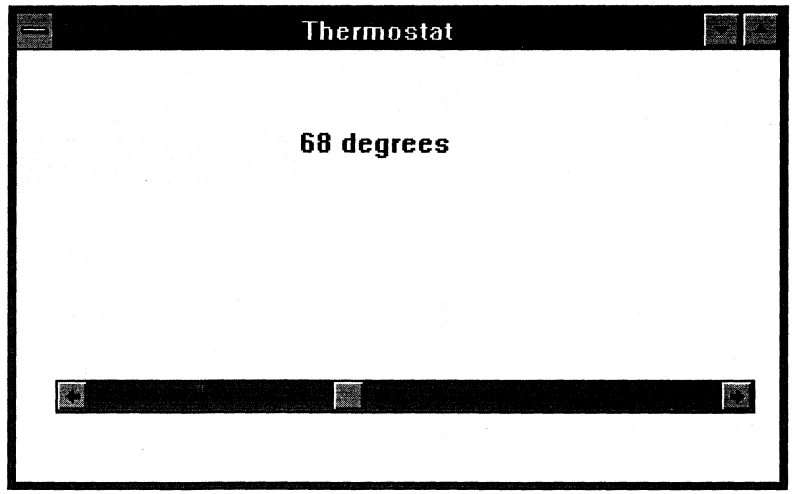


The complete file, BTNTEST.PAS, can be found on your distribution disks.

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, there may be times when you want a separate scroll bar to perform some 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 12.7 shows a typical use of a *TScrollBar* object.

Figure 12.7
A scroll bar object



Constructing scroll bars

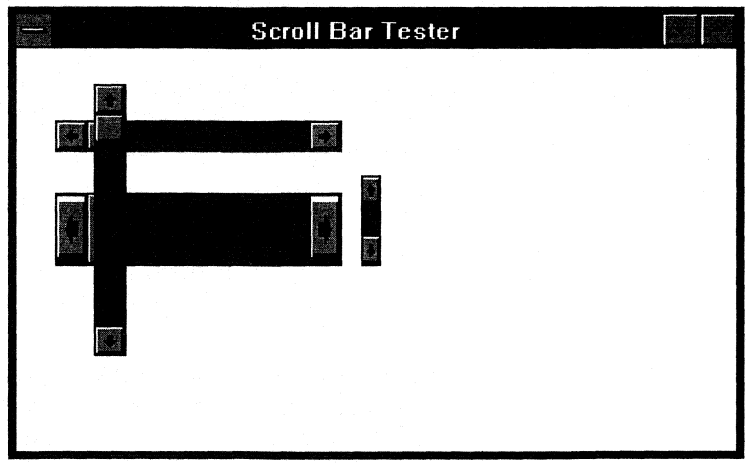
The scroll bar *Init* constructor takes a parent window, an ID, a position (*X* and *Y*), a width and a height, and a Boolean flag specifying whether the scroll bar is a horizontal scroll bar. If a width of zero is supplied for a vertical scroll bar, it will be constructed with a standard width, like that of a list box. The same is true if a height of zero is specified 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 12.7. *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 12.8 shows a variety of scroll bar objects.

Figure 12.8
A Window with a variety of scroll bars



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. Upon construction of the scroll bar object, its range is set to be 1 to 100. The thumb's "uppermost" position (top in vertical scroll bars and leftmost in horizontal scroll bars) corresponds to the position 1. Accordingly, the thumb's "lowest" position corresponds to 100. Use the *SetRange* method, described in the section, "Modifying Scroll Bars," to set the range differently.

Two other attributes of a scroll bar object are its line magnitude and page magnitude. The line magnitude, initialized to 1, is the distance, in *range* units, the thumb will move when the user clicks on the scroll bar's arrows. The page magnitude, initialized to 10, is the distance, also in range units, the thumb will move when the user clicks in the scrolling area. You can reset these values by directly manipulating a *TScrollBar*'s object fields, *LineSize* and *PageSize*.

Querying scroll bars

TScrollBar defines two methods for querying a scroll bar: *GetRange* and *GetPosition*. The *GetRange* method is a procedure that takes two integer variable arguments. The procedure fills these integers

with the uppermost and lowest thumb positions in the scroll bar's range. This method is useful when your program wants to move the thumb to its uppermost or lowest 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 seems like a job for the user of your programs, and most often it is. However, your program can also modify a scroll bar.

SetRange is a procedure that takes two integer arguments for the lowest and highest positions in the range. As a default, new scroll bars have a range of 1 to 100. You might want to change this range 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. Be sure to call *SetRange* after your scroll bar object is created:

```
procedure TParentWindowType.SetupWindow;
begin
  TWindow.SetupWindow;
  ThermScroll^.SetRange(32, 120);
end;
```

SetPosition is a procedure that takes one integer argument, the position to which you wish to move 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 bar events

When a scroll bar is scrolled, its parent window receives a scroll bar notification message. If you would like your window to respond to scrolling events, respond to the notification messages in the usual way, by defining child-ID-based response methods. However, 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 only difference you must be aware of is that the scroll bar notification codes are stored in *Msg.wParam*, rather than in *Msg.lParamHi*. Some common codes include *sb_LineUp*, *sb_LineDown*, *sb_PageUp*, *sb_PageDown*, *sb_ThumbPosition*, and *sb_ThumbTrack*. Most often, you will 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 may want to have the scroll bar object itself respond to the scroll bar notification message, building a particular response behavior into the scroll bar object. To program a scroll bar object to directly respond to its notification messages, define for its type a notify-based response method. Use the sum of

nf_First and the scroll bar notification code as the method header identifier. To do this, derive a new type from *TScrollBar*:

```
SpecializedSBar = object(TScrollBar)
  procedure SBTop(var Msg: TMessage); virtual nf_First + sb_Top;
end;

procedure TSpecializedSBar.SBTop(var Msg: TMessage);
begin
  TScrollBar.SBTop(Msg);
  { specialized handling of scroll bar going to top }
  Msg.Result := 1;
end;
```

If you don't want the scroll bar's parent window to receive the notification message after the scroll bar has processed it, set the result of the message (*Msg.Result*) to 1. If you don't set the result, the scroll bar's parent window can perform additional message processing.

Note that the method *TSpecializedSBar.SBTop* first calls *TScrollBar.SBTop*. This is the method that takes care of the physical position change of the scroll bar. *TScrollBar* has the methods *SBLineUp, SBLineDown, SBPageUp, SBPageDown, SBThumbPosition, SBThumbTrack, SBTop,* and *SBBottom* to handle the messages: (*nf_First* +) *sb_LineUp, sb_LineDown, sb_PageUp, sb_PageDown, sb_ThumbPosition, sb_ThumbTrack, sb_Top,* and *sb_Bottom,* respectively. If you want to add additional behaviors, call the specialized scroll bar's inherited method first. If you do not call the ancestor's method, your method will be responsible for moving the scroll bar.

Example: SBarTest

The *SBarTest* program creates the thermostat application shown in Figure 12.7.

The complete file, SBARTEST.PAS, can be found on your distribution disks.

Edit controls

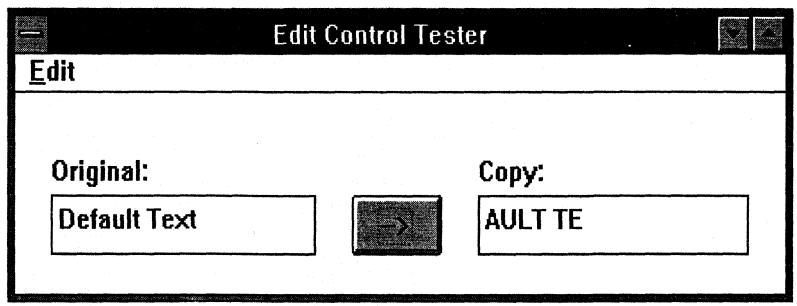
Edit controls can be described as interactive static controls. They are rectangular areas (boxed or unboxed) on the screen that can be filled with text, modified, and cleared by the user or the applica-

tion. 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 12.9 shows a window that contains two edit controls.

Figure 12.9
A window with edit controls



Constructing edit controls

The edit control's *Init* constructor takes a parent window, control ID, default (initial) text, *X*, *Y*, *W*, and *H* dimensions, a maximum text length, and a Boolean flag, *Multiline*. The edit control gets constructed with the styles *ws_Child* and *ws_Visible* (from *TControl.Init*), *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*:

```
EC1 := New(PEdit, Init(@Self, id_EC1, 'Default Text', 20, 50, 150,
  30, 40, False));
```

If *Multiline* is *True* the edit control gets the styles *es_MultiLine*, *es_AutoVScroll*, *ws_VScroll*, and *ws_HScroll*:

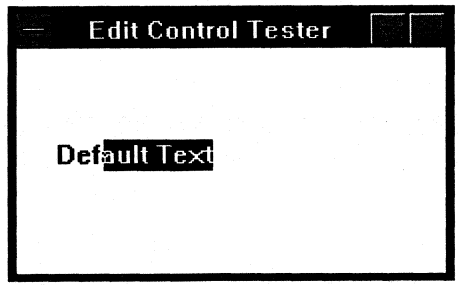
```
EC2 := New(PEdit, Init(@Self, id_EC2, '', 20, 20, 200, 150, 40,
  True));
```

After calling *Init*, the edit control's style can be changed:

```
EC3 := New(PEdit, Init(@Self, id_EC3, 'Default Text', 20, 20, 150,
  20, 40, False));
EC3^.Attr.Style := EC3^.Attr.Style and (not ws_Border);
```

These lines produce the edit control shown in Figure 12.10.

Figure 12.10
An edit control created with no border



The following table summarizes some common edit control styles:

Table 12.3
Common edit control styles

Style	Result
es_Left	Left justified
es_Center	Center justified
es_Right	Right justified
es_MultiLine	Multiline edit control
es_UpperCase	Converts to uppercase characters
es_LowerCase	Converts to lowercase characters
es_AutoVScroll	Adds automatic vertical scrolling for multiline control
es_AutoHScroll	Adds automatic horizontal scrolling
ws_Border	Adds rectangular box around edit control
ws_VScroll	Adds vertical scroll bar
ws_HScroll	Adds horizontal scroll bar

Clipboard and editing operations

You can directly transfer text between an edit control object and the Windows clipboard. Here is how to call the clipboard methods:

```
EC1^.Cut;
EC1^.Copy;
EC1^.Paste;
```

Other editing methods are available:

```
EC1^.Clear;
EC1^.DeleteSelection;
EC1^.Undo;
```

Often, you will want to give the user access to these methods through an editing menu. An edit control object will automatically respond to menu choices such as Edit | Copy and Edit |

Undo. *TEdit* defines command-based methods, such as *CMEditCopy* and *CMEditUndo*, which are invoked in response to a particular menu selection (command) in the edit control's parent window. *CMEditCopy* calls *Copy* and *CMEditUndo* calls *Undo*. The following table lists the methods invoked in response to menu selections with a particular menu ID (defined in WOBJECTS.H):

Table 12.4
Menu IDs and the methods they invoke

Menu ID	Method invoked
cm_EditCut	*Cut*
cm_EditCopy	*Copy*
cm_EditPaste	*Paste*
cm_EditClear	*Clear*
cm_EditDelete	*DeleteSelection*
cm_EditUndo	*Undo*

All you have to do to add an editing menu to a window that has edit control objects as child windows is define a menu resource for the window using the menu IDs listed here. You need not write any methods.

One additional editing method available is the Boolean function, *CanUndo*, which determines if the last change can be undone.

Implementation

The automatic edit menu response is facilitated by ObjectWindows' child window message passing mechanism, which first passes a message generated by a child window event to the child window. In the case of controls as child window, the control rarely intercepts the message and it is handled by its parent window. In the case of command messages generated by an edit menu, the message goes to the edit control that has the focus.

Querying edit controls

Often, you will 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*.

You will notice that *TEdit*'s query methods that return text from an edit control retain the text's formatting. This is important only for multiline edit controls, which allow text to appear in more than one line. In this case, returned text that spans more than one line in the edit control contains two extra characters for each line break: *carriage return* (#13) and *linefeed* (#10). When this text is inserted back in an edit control, pasted from the clipboard, written to a file, or printed to a printer, the line breaks appear as they did in the edit control. Thus, when you use a query method that gets a specified number of characters, be sure to account for the two extra characters taken up by a line break.

GetText retrieves the text from an edit control. It fills the string pointed to by the passed *PChar* argument with the contents of the edit control, including line breaks, up to the number of characters specified in the second parameter. Its return value is *False* in the case that the edit control is empty or that it holds more text than will fit into the provided string. The following procedure gets the text from an edit control and returns it in the *RetText* argument:

```
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;
```

GetLine is more specialized than *GetText*. In a multiline edit control, it returns the text in the line specified by the integer argument. Line 0 is the first line.

NumLines returns the number of lines entered in a multiline edit control. You should use *Lines* to check how many lines have been entered in an edit control before getting the text from a line with *GetLine*.

LineLength returns the number of characters in the specified line in a multiline edit control. If the line exists, but holds no characters, *LineLength* returns zero. You should use *LineLength*, *NumLines*, and *GetLine* together to safely get the text from a multiline edit control.

```
procedure TestWindow.ReturnLineText(RetText: PChar; LineNum:
  Integer);
var
  TheText: array[0..20] of Char;
begin
```

```
    RetText := nil;
    if EC1^.NumLines >= LineNum then
      if EC1^.LineLength(LineNum) < 11 then
        if EC1^.GetLine(@TheText, 20, LineNum) then
          RetText := @TheText;
end;
```

GetSubText is another method more specialized than *GetText*. It takes, as arguments, a *PChar* and two integers representing the starting and ending indexes—a range—of the edit control's text. The first character is at index zero. *GetSubText* returns the string of characters between the two indexes. In a multiline edit control, the index is counted sequentially from the first line through the remaining lines. Line breaks are counted as two characters. This lets your program reconstitute the format of the edit control's text when displaying it, printing it, transferring it to other edit controls, or placing it in the clipboard.

You might wonder where you get the indexes to pass as arguments in the *GetSubText* method call. One way is to use *GetSelection*, a method that returns the starting and ending index of the text that is currently selected, or highlighted. Usually the text has been selected by the user, but it can also be selected by the program, using *SetSelection* (see the section "Modifying Edit Controls"). The ending index returned by *GetSelection* is the index of the last selected character plus one.

You might be able to save some processing time in an application that involves processing the text from edit controls. The *IsModified* method, a Boolean function, returns *True* if the user has modified the edit control's text and *False* if it is intact. A return of *False* might save you from calling *GetText* or *GetLine*. *ClearModify* resets the change flag to *False*.

LineIndex and *GetLineFromPos* are two methods you can use to compute the placement of text in a multiline edit control only. *LineIndex* returns the number of characters (including two for line breaks) in all of the lines preceding the line specified by the integer argument. The method returns all of the edit's characters if the specified line does not exist. *GetLineFromPos* takes an index to a character position and returns the line number where that position occurs. Both of these methods are useful for querying the line structure and contents of a multiline edit control.

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.

Deleting text

Clear simply deletes the entire text entry in an edit control. You can use it as part of a resetting operation that clears some or all of the edit controls in a window. *DeleteSelection* deletes the currently selected text of an edit control. A full text editor might provide access to *Clear* and *DeleteSelection* from menu choices (see "Clipboard and editing operations"). *DeleteSelection* is a Boolean function that returns *False* if there is no text currently selected.

DeleteSubText is similar to *DeleteSelection* except that it deletes the text between the passed starting and ending positions. The removed text includes the character at the starting position but not the character at the ending position.

DeleteLine deletes all text found at the specified line. It does not, however, delete the two characters that make up the line break. Thus, no other line is affected.

Inserting text

Insert is similar to *Paste* except that it gets its inserted text from the passed argument rather than from the clipboard. It deletes any existing selected text but does not highlight the inserted text. Using *Insert*, you can implement your own text-only clipboard under program control.

SetText performs the combined actions of *Clear* and *Insert*. It deletes the entire contents of the edit control and inserts the text in the passed argument. *SetText* can also be used in a resetting operation. For example, to reset an order entry screen for an order entry system that gets most of its orders from California, you might call

```
StateField^.SetText('CA');
```

to reset the *state* entry field.

Forcing text selection and scrolling

SelectRange forces the selection, or highlighting, of text between the passed starting and ending positions, not including the character at the ending position.

Scroll forces scrolling in a multiline edit control. *Scroll* takes two integer arguments: the number of characters to scroll horizontally and the number of lines to scroll vertically. A positive integer scrolls to the right, or down, and a negative number scrolls to the left, or up. Only multiline edit controls can scroll vertically.

Sample 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 12.9.

When the user clicks on the button, the text from the left edit control (*EC1*) is copied to the right edit control (*EC2*). The text is converted to uppercase in *EC2* because it was constructed with the style *es_UpperCase*. If no text is selected in *EC1*, all of its text is copied to *EC2*. If some text is selected in *EC1*, only the selected text is copied.

The edit menu supports editing functions in whichever edit control currently has the input focus.

The complete file, EDITTEST.PAS, and its resource file, EDITTEST.RES, can be found on your distribution disks.

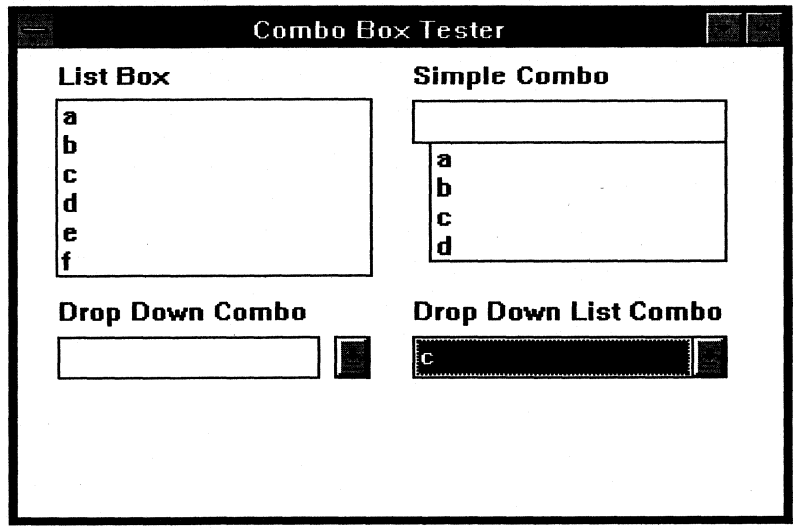
Combo boxes

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 on 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. Figure 12.11 shows the appearance of the three combo box types as well as that of a list box.

Figure 12.11
The three types of combo boxes and a list box



Simple combo boxes

All combo boxes display their edit area at all times. Some combo boxes, however, can show and hide their list box areas much like a turtle retracts its head into its shell. A simple combo box cannot hide its list box area. It is always displayed. 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 down arrow to the right of the edit area. Drop down and drop down list combo boxes are useful when you are trying to fit a lot of controls into a small area. When they are not being used, they take up much less space than a simple combo box or a list box.

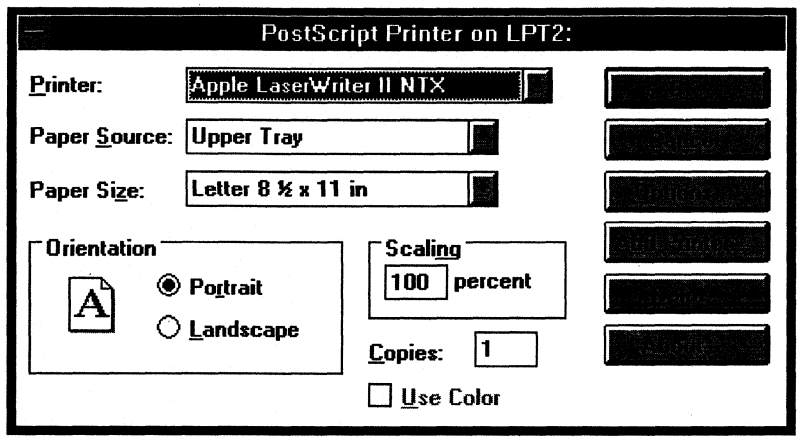
Drop down list combo boxes

The list area of a drop down list combo box behaves like the list area of drop down combo box—it appears when needed and retracts when not 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

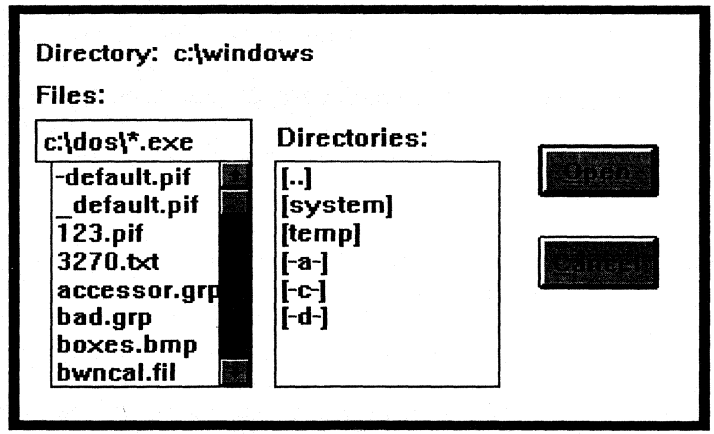
Drop down list combo boxes are useful in cases where no other selection is acceptable besides those listed in the list area. For example, when choosing printers to which to print, you can only choose a printer accessible from your system (see Figure 12.12).

Figure 12.12
A drop down list combo box



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. See Figure 12.13.

Figure 12.13
A simple combo box



Constructing combo boxes

The *Init* constructor for *TComboBox* takes a parent window, an ID, *X*, *Y*, *W*, and *H*, a style, and a maximum text length as arguments. 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 just like the corresponding parameter for an edit control, limiting the number of characters that can be entered in the edit area of the combo box. The following lines produce 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 are procedures and take no arguments. You do not need to call these methods to show and hide the list when the user clicks on the down arrow to the right of the edit area. This is an automatic mechanism of combo boxes. These methods are useful only to force the showing and hiding of the list.

Sample application: CBoxTest

The program *CBoxTest* produces the application shown in Figure 12.11. 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 on the Show and Hide buttons forces the showing and hiding of the bottom right combo box, *CB3*, by calling the methods *ShowList* and *HideList*.

The complete file, CBOXTEST.PAS, can be found on your distribution disks.

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. 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'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.

As an example, your program can bring up a modal dialog box and, after it is closed, extract information from the transfer buffer about the state of each control. Then, if the user brings up the dialog box again, the controls will be set to their states when the dialog 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 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'll have to use *InitResource* to associate objects with controls in dialog boxes or dialog windows.

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. Neither do group boxes.

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:

```
type
  SampleTransferRecord = 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: ScrollBarTransferRec;          { scroll bar range, etc. }
  end;
```

As you can see, the type of control determines the type of field defined for the transfer buffer. That's because each type of control has different information to store:

- Static controls store a character array up to the maximum length of text allowed, plus the terminating null.
- Edit controls store the edit control text buffer, up to the length defined in the *TextLen* field.
- List boxes and combo boxes store the collection of strings in the list, plus an indication of the selected item or items. For a single-selection list box, that's just an integer index to the selected string. For a multiple-selection list box, it's a record containing indexes to all the selected items. And for a combo box, it's the selected string itself.
- Check box and radio button states are stored as *Word* values, with values *bf_Unchecked*, *bf_Checked*, and *bf_Grayed* indicating those states.

- Scroll bars use another record, a *ScrollBarTransferRec*, to store the range and position of the scroll bar control. Here is the definition of *ScrollBarTransferRec*:

```
ScrollBarTransferRec = record
  LowValue: Integer;
  HighValue: Integer;
  Position: Integer;
end;
```

Defining the corresponding dialog or window

A window or dialog that uses the transfer mechanism must construct its participating control objects in the exact order in which their corresponding transfer buffer fields are defined. To enable the transfer mechanism for a window or dialog object, simply set its *TransferBuffer* field to a pointer to a transfer buffer you define.

In the case of a window with controls, construct the control objects using *Init*. For dialogs and dialog windows, use the *InitResource* constructor. For example:

```
type
  SampleTransferRecord = record
  ...
  PParentWindow = ^TParentWindow;
  TParentWindow = object(TWindow)
    TheDialog: PDialog;
    TheBuffer: SampleTransferRecord;
    ...

constructor TParentWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
var
  Stat1: PStatic;
  Edit1: PEdit;
  List1: PListBox;
  Combo1: PComboBox;
  Check1: PCheckBox;
  Radio1: PRadioButton;
  Scroll1: PScrollBar;
begin
  TWindow.Init(AParent, ATitle);
  TheDialog^.Init(@Self, PChar(101));
  New(Stat1, InitResource(TheDialog, id_Stat1));
  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));
  TheDialog^.TransferBuffer := @TheBuffer;
end;
```

In the case of a window with controls, use *Init*, rather than *InitResource*, to construct the control objects in the proper order. Another difference between windows and dialogs is that the transfer mechanism is initially disabled by default for a window's controls. To enable the mechanism, call *EnableTransfer*:

```
...
Edit1 := New(PEdit, Init(@Self, id_Edit1, "', 10, 10, 100, 30,
  40, False));
Edit1^.EnableTransfer;
...
```

To explicitly exclude a control from the transfer mechanism, call its *DisableTransfer* method after constructing it.

Transferring the data

Upon window or dialog creation (including execution of modal dialogs), data is automatically transferred from the transfer buffer to the set of participating controls.

For a modal dialog only, data is automatically transferred out of the controls and into the transfer buffer when the dialog receives a command message with a control ID of *id_OK*. Since this ID is usually returned when the user clicks an OK button to end the dialog, the dialog automatically updates its transfer buffer. Then, if the dialog is executed again, the buffer data is transferred again to the controls. Under this scheme, the dialog and the buffer remain in synch.

However, 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, supplying 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);
  TWindow.Destroy;
end;
```

Supporting transfer for custom controls

You may 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 is *tf_GetData*, copies data from the control to the location specified by the passed pointer. If the flag is *tf_SetData*, simply 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.

Transfer example

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."

The complete file, TRANTEST.PAS, can be found on your distribution disks.

C H A P T E R

13

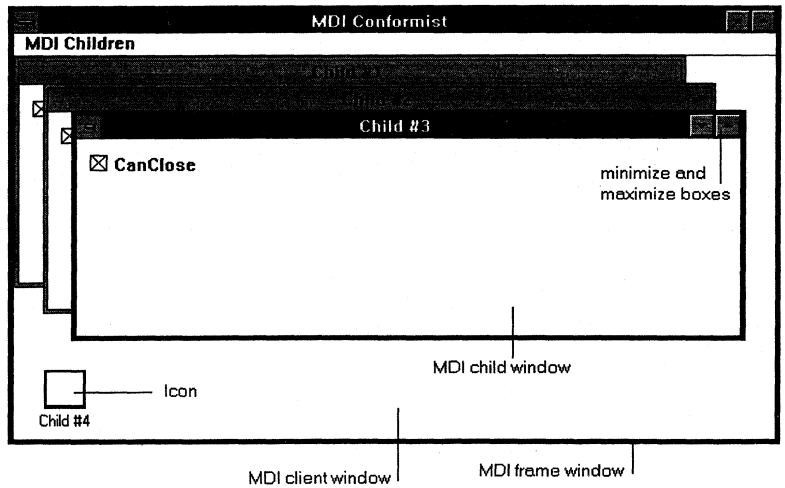
MDI objects

The multiple document interface is an interface standard for Windows applications that allow 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. In MDI-compliant applications, the user can, for example, have many open files within one application. Examples of MDI applications you might have used include Microsoft Excel, the Windows Program Manager and the Windows File Manager. The MDI standard is also part of IBM's Common User Access specification.

The components of an MDI application

There are certain components that are present in every MDI application. Most evident, there 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. It is important because it provides behind-the-scenes management of the MDI child windows.

Figure 13.1
The components of a sample MDI application



The frame window also has a menu, often labeled Window, that controls the MDI child windows with items such as Tile, Cascade, Arrange, and Close All. We will call this menu the child window menu. Each opened MDI child window is automatically appended to the end of this menu. On this menu, the currently selected window is marked with a checkmark.

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. An MDI child window cannot have a menu, so all of its 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.

Each MDI window is an object

ObjectWindows defines object types to represent MDI frame and client windows. They are *TMDIWindow* and *TMDIClient*, respectively. *TMDIWindow* descends from *TWindow*, but *TMDIClient* is actually a control and descends from *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 are instances of an object type descending from *TWindow* that you write.

TMDIWindow's methods are concerned 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 will generally descend new types from *TMDIWindow* and *TWindow* for your frame and child windows, respectively.

Constructing MDI windows

There are a few special considerations for constructing the windows of an MDI application. For a complete description of constructing windows, see Chapter 10, "Window objects."

Constructing MDI frame windows

The MDI application's frame window is also its main window, so it is constructed from within the *InitMainWindow* method of its application type. However, unlike *TWindow*, *TMDIWindow*'s *Init* constructor does not to take a parent window (frame windows, being main windows, have no parent), and it takes one more parameter, 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. However, MDI frame windows are required to have menus, so *TMDIWindow.Init* requires a menu argument and sets *Attr.Menu* for you.

The frame window's menu must include an MDI-style child window menu. This is the menu to which the MDI child window's captions will be appended during the operation of the application. One consideration is this: the frame window needs to know which top-level menu item is its child window menu. *TMDIWindow* objects store a positional integer value in the object field *ChildMenuPos*. *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 MyMDIWindow.Init(ATitle: PChar; AMenu: HMenu);
begin
  TMDIWindow.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 MDI child windows

TMDIWindow defines an automatic response method called *CreateChild* that is invoked upon a menu selection bearing the command ID *Create_Child*. Usually this menu item is called New or Create. As it is 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 (some descendant of *TWindow*), redefine *InitChild* for your MDI frame window type:

```
function MyMDIWindow.InitChild: PWindowsObject;
begin
  InitChild := New(PMyChild, Init(@Self, 'New Child Window'));
end;
```

However, you may 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 will also have to explicitly create the child window:

```
procedure MyMDIWindow.SetupWindow;
var
  ARect: TRect;
  NewChild: PMyChild;
begin
  TMDIWindow.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 will 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.

Message processing 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 it. 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.

Managing MDI 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.

Child 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 de-activates one child window by activating another. For example, the frame window's menus 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.

Child window menu

TMDIWindow defines Windows message response 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 13.1
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-based messages generated by the child window menu.

Sample MDI application

The program *MDITest* produces the MDI-compliant application pictured in Figure 13.1.

The complete file, MDITEST.PAS, can be found on your distribution disks.

PART 3

Advanced ObjectWindows

C H A P T E R

14

Memory management

Normally, you'll never have to deal with Windows' memory management facilities directly. Turbo Pascal takes care of most memory management for you. By and large, you'll simply use the normal *New* and *Dispose* (or *GetMem* and *FreeMem*) routines you would always use to allocate dynamic variables, and the compiler will then take care of making sure that the appropriate space is allocated from Windows' global heap. The ObjectWindows function *MemAlloc* can also be used, with the additional protection that it checks the safety pool. (*MemAlloc* and the safety pool are described in Chapter 19.)

There may be situations, however, when you need to deal with the memory manager directly. Those situations are outlined in the following sections, along with an overview of how to handle allocation and use of Windows memory.

Using the memory manager

While developing Turbo Pascal programs for Windows, you may encounter situations in which your program's requirements for memory exceed what the compiled program can provide directly. For example, suppose your application has a function that loads a bitmap file from disk and displays it. The program would need someplace to store the bitmap's data before handing it off to Windows to display. A bitmap's data can be quite large; too large, perhaps, to fit into a Pascal variable. To solve this problem,

Windows provides functions that allow you to allocate and manipulate memory outside of the program's own memory areas.

Because Windows is a multitasking system, several applications may use memory simultaneously. Windows manages memory in order to provide each application with the most efficient memory use possible. In this chapter, you will learn about the types of memory available in Windows, as well as what you need to know to create and access Windows memory from within your own applications.

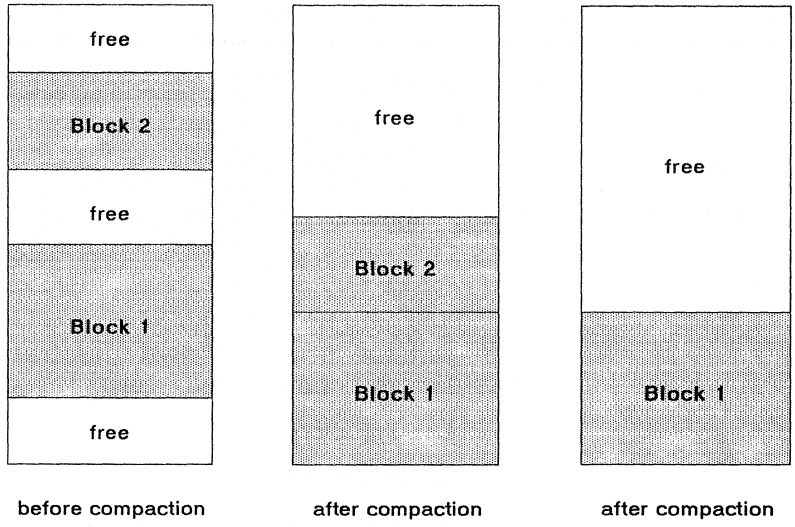
How Windows manages memory

In most DOS-based memory management systems, the memory you allocate remains fixed at a specific location during its use. In Windows, however, memory can also be *moveable* and *discardable.* A block of memory allocated as moveable has no fixed address; Windows may move it when necessary to make the best use of available memory. For example, two non-contiguous blocks of allocated memory could be moved to be contiguous, thereby leaving a larger block of free memory.

Windows memory can also be allocated as discardable. Discardable memory is similar to moveable memory in that Windows can move the block, but it can also discard it by reallocating the block to zero length. This reallocation has the effect of destroying any data contained in the block. The application must, if desired, reallocate the memory block and reload it with the correct data. This process of moving or discarding memory blocks to create a larger contiguous free memory space is called *compaction.*

Figure 14.1 shows an example of compaction. At first (left), note that there are three non-contiguous blocks of free memory separated by blocks 1 and 2. Suppose a request was made for memory of a size larger than any of the three blocks of free memory. Assume that blocks 1 and 2 are moveable. Compaction would move these blocks together, thus creating a larger contiguous area of free memory (center). Now suppose there is still not enough free memory available for the allocation. If block 2 was also discardable, it would be removed from memory (reallocated to zero length), leaving even more contiguous free memory (right).

Figure 14.1
An example of compaction



Handles and addresses

From the discussion above, it may seem that Windows memory is difficult to manage from an application's point of view. Fortunately, this is not the case. When your application allocates Windows memory, it receives a handle, not a pointer, to the memory block. This handle is not the address of a physical location in memory; it is more like a unique name that the application uses to refer to the block when making subsequent memory management function calls. You can think of a handle as a pointer to a pointer. If the handle is equal to 0, it is invalid.

Without an address, however, it would be impossible to access a moveable memory block's data. To get the address of the block, you must lock it. Locking a block returns a pointer to the beginning of the block. While a memory block is locked, the pointer you obtained during the locking procedure remains valid until the block is unlocked, even if the block is moveable or discardable. You should unlock the block when you are finished accessing its data. Failure to do so will make memory management less efficient, as Windows cannot move or discard a locked block in order to increase available free memory. Remember that once a block is unlocked, the pointer obtained by locking it is no longer valid. An invalid pointer is equal to **nil**.

Local and global memory

Windows lets your application create and use blocks of memory. This memory can come from two places, the local heap and the global heap. The global heap is all memory which is both free (not allocated) and available to all applications. Windows, applications running under Windows, and global memory blocks allocated by Windows applications all reside in the global heap. The local heap is memory available only to your application.

You must make the choice between local and global memory for each specific memory need. This choice can depend on several factors, as summarized below:

Local memory

- Faster access.
- Limited size (less than 64K bytes).

Global memory

- Blocks aligned on 32-byte boundaries.
- Twenty bytes reserved by Windows for overhead per block.
- System-wide limit on handles in Standard and 386-Enhanced modes.
- Slower access.
- More memory available.

The deciding factor is usually the amount of memory you require to carry out your task. If you need more than 1K of memory, you'd probably be better off using global memory. On the other hand, if you need small blocks of memory that are only used for a short time, local memory is the best bet.

Using the local heap

The local heap contains memory that can only be accessed by a specific instance of your application. In Windows, the local heap is usually set up in the application's data segment. This data segment also contains the stack, program global variables, and global variables of all units used by the program. All remaining space in the data segment (up to 64K) is available for use as the local heap.

Since Windows does not automatically supply your application with a local heap, you must specify a local heap size during compilation with the **$M** compiler directive. The second parameter of the **$M** compiler directive sets the size of your application's local heap (by default, 8K). See Chapter 21, "Compiler directives," in the *Programmer's Guide* for more information on using **$M**.

The maximum local heap size is also affected by whether the data segment that contains the local heap is fixed or moveable. If it is fixed, you can only allocate a local block with a maximum size equal to the heap size you specified during compilation and linking. If the data segment is moveable and a request is made for more memory beyond what you specified as the local heap size, Windows will allocate additional memory (up to the 64K limit) to try to satisfy the request. If need be, Windows can move the data segment in order to get the additional memory, thereby invalidating all long pointers to local data.

Allocating and accessing local memory

The *LocalAlloc* Windows function allocates a memory block of size and type specified as the parameters of the function call. You must decide whether the block will be fixed, moveable, or moveable and discardable (a block must be moveable to be discardable). These options, specified in the *Flags* parameter in the function call, are:

- *lmem_Fixed*, for a fixed memory block.
- *lmem_Moveable*, for a moveable memory block.
- *lmem_Moveable* **or** *lmem_Discardable*, for a discardable memory block.

When a new block is allocated in a local heap, other blocks in the local heap may be moved or discarded. To prevent these other blocks from being discarded, add *lmem_NoDiscard* to the *Flags* parameter. To prevent other blocks from being moved or discarded, add *lmem_NoCompact*.

The *LocalAlloc* function returns a handle to the memory block if the allocation is successful, and returns 0 if it's unsuccessful. If the allocation is successful, the memory block will be at least of the size requested, and it may be larger. Use the *LocalSize* function to determine its actual size. The maximum possible size of a local block is determined by the size of the local heap.

Even though you have the handle to the block, you cannot yet access its data. To do this, you need to lock the block to get its address in memory. This is done by calling the *LocalLock* function. *LocalLock* temporarily fixes the block at a constant location in the local heap, and returns a pointer to the block. The address returned by this function will remain valid until the block is unlocked, using the *LocalUnlock* function. *LocalLock* increments the lock count of the block. The lock count, also called the reference count, is a tally of the number of times the block has been fixed in memory.

You should always check the value returned from the *LocalLock* function; it is not guaranteed to be a valid pointer. The return value will be **nil** if the handle it was passed was invalid, or if the block has been discarded (the block must have been *lmem_Discardable*). The following code allocates a 256-byte block of local memory.

```
var
  LocalHandle: THandle;
  PtrLocalData: Pointer;

begin
  LocalHandle := LocalAlloc(lmem_Moveable or lmem_Discardable, 256);
  if LocalHandle <> 0 then
  begin
    PtrLocalData := LocalLock(LocalHandle);
    if PtrLocalData <> nil then
    begin
      { process data using PtrLocalData as pointer }
      LocalUnlock(LocalHandle);
    end
    else { lock failed };
  end
  else { allocation failed };
end;
```

If a local memory block was allocated with the *lmem_Fixed* attribute, it will not move in memory. Therefore, you do not have to call *LocalLock* to lock the object at a fixed address. In this case, and only in this case, the 16-bit handle returned from the *LocalAlloc* call is also the 16-bit address of the memory block.

Freeing and discarding local memory blocks

You can free local memory blocks using the *LocalFree* function. Freeing a local block removes its contents from the local heap and removes its handle from the table of valid local memory handles. This handle cannot be reused. An application should always free all memory blocks before exiting, and should free any memory block that it no longer needs.

You can explicitly discard a local memory block using the *LocalDiscard* function. Discarding a local block reallocates its size to zero, removing its contents from the local heap. Its handle, however, is still valid, and can be reused by calling the *LocalReAlloc* function with the handle and a new size value. You can save some processing time in your application by discarding and reallocating handles instead of freeing and newly allocating them.

```
var
  LocalHandle: THandle;
begin
  { LocalHandle previously allocated }
  LocalDiscard(LocalHandle);
  { now reuse the handle to create a new memory block }
  LocalHandle := LocalReAlloc(LocalHandle, 256, lmem_Moveable);
  ...
end;
```

A local memory block cannot be freed or discarded unless its lock count is zero.

Reallocating and modifying local memory blocks

You can change the size of a block while preserving its contents by using the *LocalReAlloc* function. Windows will truncate the block if you specify a new size smaller than the current size. If you specify a new size larger than the current size, the new area will contain zeroes if you specify *lmem_ZeroInit*; otherwise it will contain undefined data. Just as with *LocalAlloc*, existing local blocks can be moved or discarded by Windows during reallocation of a block using *LocalReAlloc*. You can specify *lmem_NoDiscard* or *lmem_NoCompact* to prevent discarding or moving of other blocks during reallocation.

LocalReAlloc can also be used to change the block's attributes between *lmem_Moveable* and *lmem_Discardable*. You must specify

lmem_Modify in addition to the new attribute. You cannot use *LocalReAlloc* with the *lmem_Modify* attribute to change the block to or from *lmem_Fixed*. This code changes the type of a block to be discardable:

```
var
  LocalHandle: THandle;

begin
  LocalHandle := LocalAlloc(lmem_Moveable, 256);
  LocalHandle := LocalReAlloc(localHandle, 256, lmem_Modify or
    lmem_Discardable);
end;
```

Querying local memory blocks

The *LocalSize* and *LocalFlags* functions can be used to obtain information about local memory blocks. *LocalSize* returns the size of the local block. Since memory allocation can create a block larger than the size requested, you should use *LocalSize* to determine the actual size of the block, if necessary.

```
var
  LocalHandle: THandle;
  BlockSize: Word;

begin
  LocalHandle := LocalAlloc(lmem_Moveable, 256);
  if LocalHandle <> 0 then BlockSize := LocalSize(LocalHandle);
end;
```

LocalFlags is used to determine the lock count of the memory block, as well as whether the block is discardable and, if so, whether the block has been discarded.

```
var
  LocalHandle: THandle;
  Flags, LockCount: Word;

begin
  { assuming the block has been allocated and maybe even locked }
  Flags := LocalFlags(LocalHandle);
  LockCount := Flags and lmem_LockCount;
end;
```

You can use the *LocalCompact* function to obtain the size of the largest block of free memory in the local heap; simply specify zero as the parameter.

Programming considerations

You should always unlock your moveable and discardable memory blocks as soon as you have finished accessing them. This is especially true if you will be allocating other blocks in the local heap. If you do not unlock these blocks, Windows cannot move them around to create sufficient free memory for future allocations. You should also free all memory blocks before exiting your program.

Using the global heap

The global heap is the system-wide memory that is shared by Windows and applications. The amount of global heap available to your application depends on the mode that Windows is running in. Because global memory is shared, your application must attempt to use it as efficiently as it can, or total system performance could suffer.

Allocating and accessing global memory

The *GlobalAlloc* function allocates a memory block of size and type specified as the parameters of the function call. As with local memory allocation, you must decide whether the block will be fixed, moveable, or moveable and discardable (a block must be moveable to be discardable). These options, specified in the *Flags* parameter, are:

- *gmem_Fixed*, for a fixed memory block.
- *gmem_Moveable*, for a moveable memory block.
- *gmem_Moveable* **or** *gmem_Discardable*, for a discardable memory block.

When a block is allocated in a global heap, other blocks in the global heap may be moved or discarded. To prevent other blocks from being discarded, add *gmem_NoDiscard* to the *Flags* parameter. To prevent blocks from being moved or discarded, add *gmem_NoCompact*.

The *GlobalAlloc* function returns a handle to the memory block if the allocation was successful, or 0 if it was unsuccessful. The value returned by *GlobalAlloc* should always be checked to

determine if the allocation was successful. If the allocation is successful, the memory block will be at least of the size requested, and it may be larger. Use the *GlobalSize* function (described below) to determine its actual size. The largest global memory block that can be allocated in Windows Standard Mode is 1 MB; in 386-Enhanced Mode, it is 64 MB.

To obtain the address of a global memory block, you must lock the block. This is done by calling the *GlobalLock* function. In real mode, *GlobalLock* temporarily fixes the block at a constant location in the global heap; in other modes, the block is not fixed. In all modes, however, *GlobalLock* returns a far pointer that will remain valid until the block is unlocked using *GlobalUnlock*.

You should always check the value returned from the *GlobalLock* function; it is not guaranteed to be a valid pointer. The return value will be **nil** if the handle it was passed was invalid, or if the block has been discarded (the block must have been allocated as *gmem_Discardable*). This code allocates a global memory block:

```
type
  PBigArray = ^TBigArray;
  TBigArray = array[0..20000] of Byte;
var
  GlobalHandle: THandle;
  PtrGlobalData: PBigArray;
  Value: Byte;

begin
  GlobalHandle := GlobalAlloc(gmem_Moveable, 20000);
  if GlobalHandle <> 0 then
  begin
    PtrGlobalData := GlobalLock(GlobalHandle);
    if PtrGlobalData <> nil then
    begin
      { process data using PtrGlobalData as pointer }
      { a very simple example }
      PtrGlobalData^[0] := 255;
      Value := PtrGlobalData^[0];
      GlobalUnlock(GlobalHandle);
    end
    else { the lock failed };
  end
  else { allocation failed };
end;
```

In Real mode, *GlobalLock* must fix the block in memory until it is unlocked. The lock count is thus increased by one, indicating that

the memory has actually been fixed, and that it cannot be freed or discarded until it is unlocked. Calling *GlobalUnlock* decrements the lock count by one. In Windows Standard and 386-Enhanced modes, unless the block is discardable, the lock count is not increased because the block is not actually fixed in memory. The pointer will always be valid even though Windows may move the block in memory.

Unlike local memory blocks, if a global block was allocated with the *gmem_Fixed* attribute, you must still lock the block before accessing its data. The handle returned by *GlobalAlloc* is not a pointer to the block in this case.

Other ways of locking global memory blocks

There are several additional functions that may be used to lock and unlock global memory. *GlobalFix* locks the block in memory and prevents it from moving, regardless of what mode Windows is running in. Because the block is actually fixed, its lock count is increased by one. Don't use *GlobalFix* unless it's absolutely necessary, as it can interfere with Windows memory compaction. Few applications require memory to be fixed in this way because the pointer remains valid even if the block has been moved.

GlobalUnfix unlocks the block and decrements its lock count.

GlobalWire moves a segment into low memory and locks it, increasing the lock count by one. *GlobalWire* is useful for memory blocks that must be locked for long periods. Because the memory block is placed in low memory, it will not interfere with Windows memory compaction while fixed. *GlobalUnWire* unlocks the block and decrements the lock count.

Freeing and discarding global memory blocks

Global memory blocks may be freed and discarded in the same manner as local blocks; the only difference is in the function names, *GlobalFree* and *GlobalDiscard*.

Reallocating and modifying global memory blocks

You can change the size of a block while preserving its contents by using the *GlobalReAlloc* function. Windows will truncate the block if you specify a size smaller than the current size. If you specify a size larger than the current size, the new area will

contain zeroes if you specify *gmem_ZeroInit*; otherwise it will contain undefined data. Just as with *GlobalAlloc*, existing global blocks can be moved or discarded by Windows during reallocation of a block using *GlobalReAlloc*. You can specify *gmem_NoDiscard* or *gmem_NoCompact* to prevent discarding or moving of blocks during reallocation.

GlobalReAlloc can also be used to change the block's attributes between *gmem_Moveable* and *gmem_Discardable*. You must specify *gmem_Modify* in addition to the new attribute. You can also use *GlobalReAlloc* with the *gmem_Modify* attribute to change the global block to or from *gmem_Fixed*. Note that you could not do this for local blocks with *LocalReAlloc*.

If you are reallocating a memory block to a new size that crosses a multiple of 64K, Windows may return a new handle for that block. If your application is running in standard mode, this applies for reallocation of a block past multiples of 65,519 bytes. Because of this, you must preserve the old handle and check the new handle returned from the *GlobalReAlloc*. If the *GlobalReAlloc* call is successful, you must make the old handle equal to the new handle; otherwise you should preserve the old handle:

```
var
  GlobalHandle: THandle;
  TempHandle: THandle;

begin
  { TempHandle originally allocated for 32K }
  { now reallocate past a 64K boundary }
  TempHandle := GlobalReAlloc(GlobalHandle, 102400),
    gmem_Moveable);
  if TempHandle <> 0 then
    GlobalHandle := TempHandle; { in case it changed }
  else { block was not reallocated }
end;
```

Querying global memory blocks

The *GlobalSize* and *GlobalFlags* functions can be used to obtain information about global memory blocks. *GlobalSize* returns the size of the global block. *GlobalFlags* is used to determine the lock count of the memory block, as well as whether the block is discardable and, if so, whether the block has been discarded. *GlobalFlags* also indicates whether the block was allocated as *gmem_DDEShare* or *gmem_Not_Banked*.

```
var
  globalHandle: THandle;
  flags: Word;
begin
  Flags := GlobalFlags(globalHandle);
  { test if block is discardable }
  if Flags and gmem_Discardable then { block is discardable }
end;
```

The *GlobalCompact* function can be used to obtain the size of the largest block of free memory in the global heap; simply specify zero as the parameter. If you specify zero, no compaction actually takes place. The value returned is the largest free block if compaction actually did take place. You can also use *GlobalCompact* to forcibly discard all unlocked discardable blocks in the system by specifying –1 as the parameter.

```
var
  globalHandle: THandle;
  largestSize: Longint;
begin
  largestSize := GlobalCompact(Long(0));
  GlobalCompact(Long(-1));    { forcibly discard all unlocked blocks }
end;
```

Changing global discarding

Windows uses a least-recently-used (LRU) algorithm to determine which discardable block should be discarded when necessary. You can specify where a certain memory block resides in the to-be-discarded list by using the *GlobalLRUOldest* and *GlobalLRUNewest* functions. Calling *GlobalLRUOldest* will indicate to Windows that the block specified as a parameter to the function should be the next block to discard. *GlobalLRUNewest* moves the specified block to the end of the discard list; it will be discarded last.

Receiving low-memory warnings

When Windows determines that it is spending too much time compacting memory, it will send a *wm_Compacting* message to all top-level windows of applications currently running. This message indicates that system memory is low, and that each application should free as much global memory as possible immediately.

Programming considerations

It is always a good practice to unlock your global memory blocks as soon as you have finished accessing them. If you do not unlock these blocks, Windows cannot move them around to create sufficient free memory for future allocations. You should also free all memory blocks before exiting your program.

If you decide to use global memory, you should avoid allocating small memory blocks (less than 128 bytes) in the global heap, due to the overhead and byte-alignment of global memory. You should also refrain from allocating large numbers of global memory blocks. A better practice is to combine them into a smaller number of larger blocks.

Never pass ordinary global memory handles between applications to create a shared memory area. Use only DDE and the clipboard to provide shared memory. See Chapter 16, "Dynamic Data Exchange."

C H A P T E R

15

Dynamic-link libraries

Dynamic-link libraries (DLLs) are executable modules of code and/or data that can be linked to an application at run time. DLLs allow an application developer to develop and maintain individual components of an application, in effect, adding features without modifying the compiled application. DLLs offer ready-made units of functionality, such as file format filters, that can be plugged into an application that is aware of it. DLL modules use memory efficiently because they can be shared by many concurrently executing applications.

The intricate details of DLL structure and use can be found in Chapter 10, "Dynamic-link libraries," in the *Programmer's Guide*. This chapter focuses on using DLLs in Windows applications.

Accessing DLL routines

A DLL consists of a library of functions, and can be written in a variety of programming languages, independent of the language used to access the DLL. Declaring a DLL function from within a Turbo Pascal program is similar to declaring an external routine.

```
function MyExternalRoutine(Parameter1: Word): Word; far;
external 'MYDLL' index 12;
```

The above function definition declares *MyExternalRoutine* to be a function in the external DLL module, MYDLL, with an ordinal value of 12. If the specified module is not already loaded when

the application declaring this external routine is loaded, Windows will load the specified modules into memory. Since linking is performed at run time, the ordinal value defines an offset value into the module, used to locate the routine. Most commercially-distributed DLLs will provide you with the ordinal values of their routines.

A simple DLL example

Here is an example of a dynamic-link library to handle some simple mathematical calculations. It consists of three files:

- MATHDLL.PAS, a library that defines the calculation routines.
- MATH.PAS, a unit that defines the interface to *MathDLL*.
- DLLTEST.PAS, a main program that uses the *Math* unit.

These files are on your distribution disks, and can be loaded and tested from the IDE.

This is MATHDLL.PAS.

```
library MathDll;

uses WinTypes, WinProcs;

function Power(X, Y: Real): Real; export;
begin
  Power := Exp(Y * ln(X));
end;

function Payments(Period, Interest, Term, Principal: Real): Real;
  export;
begin
  Payments := (Principal * Interest / Period) / (1 - Power(1 +
    Interest / Period, -Term * Period));
end;

function Principals(Payment, Period, Interest, Term: Real): Real;
export;
begin
  Principals := Payment * ((1 - Power(1 + Interest / Period,
    -Term * Period)) / (Interest / Period));
end;

procedure WriteError(Window: HWnd; ErrorMessage: PChar); export;
var S: String;
begin
  MessageBox(Window, ErrorMessage, 'Error', mb_OK);
end;

exports Power        index 1;
```

```
exports Payments     index 2;
exports Principals   index 3;
exports WriteError   index 4;

begin
end.
```

In order to use these library routines in a Turbo Pascal program, you need to create a unit that imports the routines from the DLL. Here is a unit that makes all four routines in *MathDLL* available:

This is MATH.PAS.

```
unit Math;

interface

function Power(X, Y: Real): Real;
function Payments(Period, Interest, Term, Principal: Real): Real;
function Principals(Payment, Period, Interest, Term: Real): Real;
procedure WriteError(Window: Word; ErrorMessage: PChar);

implementation

function Power;       external 'MathDll' index 1;
function Payments;    external 'MathDll' index 2;
function Principals;  external 'MathDll' index 3;
procedure WriteError; external 'MathDll' index 4;
end.
```

Note that all the routines are implicitly **far** because they are listed in the interface section of the unit.

To make use of the library functions, all your application needs to do is include *Math* in its **uses** clause. Your application need not be aware that the routines being called are actually in a library; that part is transparent.

For example, the following program uses the *Math* unit to perform some calculations:

This is DLLTEST.PAS.

```
program DLLTest;

{$R MATH}

uses WinTypes, WinProcs, WObjects, Math;

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

  TPaymentDlg = object(TDialog)
    Period, Interest, Term, Principal : Real;
    function LoadFields: Boolean;
    procedure OK(var Msg: TMessage); virtual id_First + id_Ok;
```

```
  end;

  TPrincipalDlg = object(TDialog)
    Payment, Period, Interest, Term: Real;
    function LoadFields: Boolean;
    procedure OK(var Msg: TMessage); virtual id_First + id_Ok;
  end;

  PMyWindow = ^TMyWindow;
  TMyWindow = object(TWindow)
    PaymentDlg: TPaymentDlg;
    PrincipalDlg: TPrincipalDlg;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    procedure Payment(var Msg: TMessage); virtual cm_First + 201;
    procedure Principal(var Msg: TMessage); virtual cm_First + 202;
  end;

function TPaymentDlg.LoadFields: Boolean;
var
  E: Integer;
  S: String;
begin
  LoadFields := False;
  S[0] := Char(GetDlgItemText(HWindow, 104, @S[1], 20));
  Val(S, Interest, E);
  if E <> 0 then Exit;
  S[0] := Char(GetDlgItemText(HWindow, 105, @S[1], 20));
  Val(S, Term, E);
  if E <> 0 then Exit;
  S[0] := Char(GetDlgItemText(HWindow, 112, @S[1], 20));
  Val(S, Period, E);
  if E <> 0 then Exit;
  S[0] := Char(GetDlgItemText(HWindow, 106, @S[1], 20));
  Val(S, Principal, E);
  if E <> 0 then Exit;
  LoadFields := True;
end;

procedure TPaymentDlg.Ok(var Msg: TMessage);
var S: String;
begin
  if not LoadFields then
    WriteError(HWindow, 'All fields must have values')
  else
  begin
    Str(Payments(Period, Interest, Term, Principal):10:2, S);
    S := S + #0;
    while S[1] = ' ' do Delete(S, 1, 1);
    SetDlgItemText(HWindow, 113, @S[1]);
  end;
end;
```

```
function TPrincipalDlg.LoadFields: Boolean;
var
  E: Integer;
  S: String;
begin
  LoadFields := False;
  S[0] := Char(GetDlgItemText(HWindow, 104, @S[1], 20));
  Val(S, Interest, E);
  if E <> 0 then Exit;
  S[0] := Char(GetDlgItemText(HWindow, 105, @S[1], 20));
  Val(S, Term, E);
  if E <> 0 then Exit;
  S[0] := Char(GetDlgItemText(HWindow, 112, @S[1], 20));
  Val(S, Period, E);
  if E <> 0 then Exit;
  S[0] := Char(GetDlgItemText(HWindow, 106, @S[1], 20));
  Val(S, Payment, E);
  if E <> 0 then Exit;
  LoadFields := True;
end;

procedure TPrincipalDlg.Ok(var Msg: TMessage);
var S: String;
begin
  if not LoadFields then
    WriteError(HWindow, 'All fields must have values')
  else
  begin
    Str(Principals(Payment, Period, Interest, Term):10:2, S);
    S := S + #0;
    while S[1] = ' ' do Delete(S, 1, 1);
    SetDlgItemText(HWindow, 113, @S[1]);
  end;
end;

constructor TMyWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, 'Menu');
end;

procedure TMyWindow.Payment(var Msg: TMessage);
begin
  PaymentDlg.Init(@Self, 'PaymentDlg');
  PaymentDlg.Execute;
  PaymentDlg.Done;
end;

procedure TMyWindow.Principal(var Msg: TMessage);
begin
  PrincipalDlg.Init(@Self, 'PrincipalDlg');
```

```
  PrincipalDlg.Execute;
  PrincipalDlg.Done;
end;

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

var MyApplication: TMyApplication;
begin
  MyApplication.Init('Hello');
  MyApplication.Run;
  MyApplication.Done;
end.
```

C H A P T E R

16

Dynamic Data Exchange

Because Windows permits many applications to run concurrently, it is often desirable for these applications to share data at run time. To support inter-application data transfer, Windows supplies a clipboard, a customizable dynamic data exchange (DDE) protocol, and the ability to define new Windows messages and send them among applications.

The Windows clipboard

The Windows clipboard acts as a repository for data in a variety of formats. Each Windows application that places data into the clipboard can select a predefined format, or define one of its own. An application that retrieves data from the clipboard can query the clipboard to find out about the formats of the currently stored data. An application can simultaneously place data in the clipboard in a variety of formats, offering a choice of formats to the application that retrieves the data from the clipboard.

Clipboard formats

Windows' predefined clipboard formats are specified in calls to Windows clipboard functions by *cf_* constants. They include:

Table 16.1
Clipboard formats

Format	Meaning
cf_Text	Null-terminated array of characters.
cf_Bitmap	Windows bitmap format.
cf_Sylk	Microsoft symbolic link format.
cf_Tiff	Tag image file format
cf_MetafilePict	Windows metafile picture format

These formats are public standards supported by many existing Windows applications. To use a specialized, private format, or to establish a new standard, you might want to register your own clipboard formats. Data placed in the clipboard with your own formats can only be retrieved by applications that are aware of your format.

Each application that intends to use a new format should call the *RegisterClipboardFormat* function to register the new format. The first call to *RegisterClipboardFormat* in a Windows session returns a new, unique clipboard format identifier, similar to the *cf_* constants. Every additional application that registers a format of the same name is given the same format identifier. In this way, all applications will use the same identifier for this new data format.

Placing data in the clipboard

Placing data in the clipboard involves these four steps, performed by the application that wishes to place the data:

1. Open the clipboard (*OpenClipboard*).
2. Empty the clipboard (*EmptyClipboard*).
3. Set the new data in the clipboard (*SetClipboardData*).
4. Close the clipboard (*CloseClipboard*).

The call to *OpenClipboard* takes, as an argument, the handle of a window. This ensures that only one window is manipulating the clipboard at one time.

When placing data in the clipboard, the application must copy the data into global memory (see Chapter 14), and pass a handle to that data in a call to the *SetClipboardData* Windows function. Once that handle has been passed, the clipboard retains ownership of the data, so the application can no longer use it. For this reason, be sure to pass a handle to a copy of your data. The *SetClipboardData* function takes, as an argument, a clipboard format identifier such as *cf_Text*. Your application can call

SetClipboardData many times, each time adding data in a different format.

The following code illustrates the procedure for placing data in the clipboard in *cf_Text* format.

```
function MyWindow.CopyText(TextString: PChar): Boolean;
var
  StringGlobalHandle: Handle;
  StringGlobalPtr: PChar;
begin
  CopyText := False;
  StringGlobalHandle := GlobalAlloc(gmem_Moveable, StrLen(TextString)
    + 1);
  if StringGlobalHandle <> 0 then
  begin
    StringGlobalPtr := GlobalLock(StringGlobalHandle);
    if StringGlobalPtr <> nil then
    begin
      StrCopy(StringGlobalPtr, TextString);
      GlobalUnlock(StringGlobalHandle);
      if OpenClipboard(HWindow) <> 0 then
      begin
        EmptyClipboard;
        SetClipboardData(cf_Text, StringGlobalHandle);
        CloseClipboard;
        CopyText := True;
      end
      else GlobalFree(StringGlobalHandle);
    end
    else GlobalFree(StringGlobalHandle);
  end;
end;
```

Notice that the application's text data, passed to *CopyText* as the *TextString* argument, is copied with *StrCopy* to *StringGlobalPtr*. Then the data stored in *StringGlobalPtr* is passed in *SetClipboardData* so that *TextString* can still be used by the application.

The global memory allocated with *GlobalAlloc* should not be allocated with the *gmem_Discardable* flag, since the clipboard (which will eventually own the global handle) will not know how to recreate the data.

Retrieving data from the clipboard

Retrieving data from the clipboard involves these four steps, performed by the application that wishes to retrieve the data:

1. Open the clipboard (*OpenClipboard*).
2. Inquire about the current clipboard formats (*IsClipboardFormatAvailable*).
3. Retrieve data from the clipboard (*GetClipboardData*).
4. Close the clipboard (*CloseClipboard*).

As with placing data into the clipboard, retrieving data from the clipboard deals with a handle to global memory. Since the clipboard owns and will continue to own its data, your application must copy the data associated with the handle. The application should not use and manipulate the handle directly.

The following function illustrates the procedure for retrieving text data from the clipboard.

```
function MyWindow.PasteText(TextString: PChar; TextSize: Integer):
Integer;
var
  StringGlobalHandle: Handle;
  StringGlobalSize: Longint;
  StringGlobalPtr: PChar;
begin
  PasteText := -1;
  if OpenClipboard(HWindow) <> 0 then
  begin
    if IsClipboardFormatAvailable(cf_Text) <> 0 then
    begin
      StringGlobalHandle := GetClipboardData(cf_Text);
      if StringGlobalHandle <> 0 then
      begin
        StringGlobalSize := GlobalSize(StringGlobalHandle);
        StringGlobalPtr := GlobalLock(StringGlobalHandle);
        if StringGlobalPtr <> nil then
        begin
          if TextSize < StringGlobalSize then
            StringGlobalSize := TextSize;
          StrLCopy(TextString, StringGlobalPtr, StringGlobalSize);
          GlobalUnlock(StringGlobalHandle);
          PasteText := StrLen(TextString);
        end;
      end;
```

```
      end;
      CloseClipboard;
    end;
  end;
```

If your application can handle multiple data formats, it should ask for the most informative data format first. For instance, if the clipboard supports a binary data format for passing data, a program should ask for it before asking for the *cf_Text* data format—there might be rounding errors introduced when converting from numbers to text and back again.

Delayed rendering

Applications that support many data formats would be significantly burdened if they had to render data in each format every time they placed data in the clipboard. As an alternative, you can delay the rendering of the data until another application requests it. In that case, it is used only when needed.

To delay rendering, pass a 0 data handle in the call to *SetClipboardData* for all formats which you wish to delay rendering. However, the application that delays rendering must preserve the last data placed in the clipboard to be rendered later. When another application requests data in one of the delay-rendered formats, Windows sends a *wm_RenderFormat* message to the application that posted the delay-rendered format. The *wm_RenderFormat's* word argument contains the requested format for the data. The application should then place the data into the clipboard in the requested format, following the previous instructions, with the exception of emptying the clipboard.

The rendering application will receive a *wm_DestroyClipboard* message when the clipboard contents are next emptied by another application. Then the application can get rid of the data it saved for the delayed rendering. If a delayed-rendering application is being destroyed, Windows sends it a *wm_RenderAllFormats* message to allow it to place its data in all the delay-rendered formats.

Inter-application messaging

Most messages a window or application receives are sent by Windows or by another window in the same application (see

user-defined messages in Chapter 7). However, it is possible for one application to send a message to another application. While this process is formalized in the Dynamic Data Exchange (DDE) protocol, Windows offers a mechanism for generalized inter-application messaging. This mechanism is most appropriate for sending notification messages or for passing simple data values. DDE is more appropriate for passing complex data.

For example, say you have written a series of workgroup applications, including an e-mail, scheduling, and calendar program. The e-mail application includes special routines to intercept e-mail notification messages from the network server. The e-mail program must then notify the scheduler and calendar programs that an e-mail message has arrived. In this case, you can define a new message, *wm_EMailArrived*, for example. Every application that will send or receive this new message must register it using the Windows function *RegisterWindowMessage*. In order to send a message from one application to another, use *SendMessage* or *PostMessage*. In order to send the message to every open window, pass *HWnd(-1)* as the window handle parameter. To verify receipt of the message, the receiving applications can send back another customized message.

However, do not pass, in the *Longint* parameter (*lParam*), a pointer to data because Windows can move your application and invalidate the pointer. Do not pass a global memory handle because the memory block could be de-allocated, moved or discarded. Problems could also arise if the sending or receiving application does not de-allocate the memory handle. In general, you should pass information directly in the *Word* or *Longint* parameters.

Dynamic data exchange

Dynamic Data Exchange (DDE) is a protocol by which two cooperating applications agree to share data. The important word is "cooperating," because the DDE protocol requires that the two applications know how to request data on the same application-defined topics. DDE is implemented as a group of Windows messages (starting with *wm_DDE_*) which, when sent from one application's window to that of another application, can pass data or make requests of the other application.

The DDE protocol also insulates both applications from the problems inherent in passing global data between applications. Instead, it uses global *atoms* for passing text strings and uses global memory, allocated with the flag *gmem_DDEShare*, for passing data. Atoms are global string constants accessible by all currently running Windows applications.

Terms

When two applications agree to exchange data, they are engaged in a conversation. A single window should engage in only one conversation at a time. An application that wishes to engage in multiple, simultaneous conversations should create multiple windows, each of which engages in one conversation. If need be, these additional windows can be hidden. The application window that initiates the conversation is called the client window and the application window that responds to this initiation is called the server.

The DDE protocol has a three-tier system for the specification of the exchanged data. At the highest level is the application name, followed by the conversation topic and then the item name. The application name is important because each instance of an application usually "understands" a particular set of application topics. For instance, a Windows word processor might use document file names as conversation topics. The item is the specific piece of data that is requested. In a word processor, the item might be the current paragraph or font.

The application and topic are used to establish the conversation and the item is the unit of data transmission. The format of the data exchanged can be any of the clipboard formats (see Table 16.1).

Establishing a conversation

A client initiates a conversation with the server, which continues until either the client or server terminates it. The messages governing conversation initiation and termination are *wm_DDE_Initiate* and *wm_DDE_Terminate*, respectively.

A client initiates a conversation by broadcasting a *wm_DDE_Initiate* message, using the *SendMessage* function, and specifying *HWnd(-1)* as the window handle parameter. Because the DDE protocol is asynchronous, only *wm_DDE_Initiate* needs

to be sent with *SendMessage*. After a conversation is established with a particular window, the *PostMessage* function is used to pass subsequent DDE messages. *PostMessage* differs from *SendMessage* in that it allows the sending application to continue running without waiting for a response. The disadvantage is that the client application is then responsible for polling for any expected responses.

More than one server application window might respond to a *wm_DDE_Initiate*; the client is responsible for selecting one server and sending *wm_DDE_Terminate* messages to the others. This way there is only one server responding to one client.

When initiating a conversation with *wm_DDE_Initiate*, an application specifies its desired application name and conversation topic. If a null value is supplied for the application name, a conversation might be started with any running application. If the supplied topic is null, a conversation on any topic might commence.

The following method can be used by a window to initiate a conversation. It takes, as arguments, an application name and conversation topic. If you wish to use null values, supply **nil** as these arguments.

```
procedure TClientWindow.InitiateConversation(AppName, TopicName:
PChar);
var
  AppGlobalAtom, TopicGlobalAtom: Word;
  lParam: Longint;
begin
  if AppName <> nil then
    AppGlobalAtom := GlobalAddAtom(AppName)
  else AppGlobalAtom := 0;
  if TopicName <> nil then
    TopicGlobalAtom := GlobalAddAtom(TopicName)
  else TopicGlobalAtom := 0;
  lParam := AppGlobalAtom or (TopicGlobalAtom shl 16);
  SendMessage(Word(-1), wm_DDE_Initiate, HWindow, lParam);
  if AppGlobalAtom <> 0 then GlobalDeleteAtom(AppGlobalAtom);
  if TopicGlobalAtom <> 0 then GlobalDeleteAtom(TopicGlobalAtom);
end;
```

Notice that in the *SendMessage* function, global atoms are passed instead of *Longint* pointers. The atoms are created only if the application or topic string are not **nil**. The application must always free up any global atoms created after the *SendMessage* call. It is safe do this because the *SendMessage* call will not return

until after all application windows in the system get a chance to handle the *wm_DDE_Initiate* message. Rules for creating and destroying global atoms will be discussed below under the appropriate DDE messages. All application windows that can handle the passed application and topic respond by sending an acknowledgement in the form of a *wm_DDE_Ack* message.

To respond to a *wm_DDE_Initiate* message, an application's window object can define an automatic message response method to be invoked in response to *wm_DDE_Initiate*. This method would check the specified topic and, if it was interested in conversing, would call another method to send acknowledgement to the client window. If the topic is a null value, the server window can choose which of its topics it might want to converse about. Either way, the server application usually creates a new window to manage the conversation. (Remember, it's one window per DDE conversation). A handle to this new window should be passed to the method that sends acknowledgement.

Here is how a server window, named *TestServer*, might respond to a *wm_DDE_Initiate* message:

```
procedure TServerWindow.ACKTopic(ClientHWnd, ConversationHWnd: HWnd;
TopicName: PChar);
var
  AppGlobalAtom, TopicGlobalAtom: Word;
  lParam: Longint;
begin
  AppGlobalAtom := GlobalAddAtom('TestServer');
  TopicGlobalAtom := GlobalAddAtom(TopicName);
  lParam := AppGlobalAtom or (TopicGlobalAtom shl 16);
  if SendMessage(ClientHWnd, wm_DDE_Ack, ConversationHWnd,
    lParam) = 0
    then
  begin
    GlobalDeleteAtom(AppGlobalAtom);
    GlobalDeleteAtom(TopicGlobalAtom);
  end;
end;
```

The window that is handling the DDE conversation as the server is passed in as a parameter, *ConversationHWnd*, instead of using the *HWindow* instance variable of the *TServerWindow* window object. This is to enforce the idea that a new window object must be created for each topic as well. The client then has the option of choosing which topic to continue the conversation with. Also notice that the global atoms are created again and that they are

deleted only if *SendMessage* fails. This is because the client, upon receiving the *wm_DDE_Ack* message, has the responsibility to delete the global atoms.

From this point on, the conversation is established. The server knows the client because it received the client's window handle from the *wm_DDE_Initiate* message and the client knows the server because it received the server's window handle from the *wm_DDE_Ack* message. The window handles are then used in all subsequent DDE messages between the two applications.

Terminating a conversation

Either the client or the server window can terminate a current conversation by sending a *wm_DDE_Terminate* message. Usually the client sends the *wm_DDE_Terminate* but the server might send this message if its application is closing. Upon receipt of a *wm_DDE_Terminate,* the receiving window must send a *wm_DDE_Terminate* message back. Any DDE messages received after a *wm_DDE_Terminate* message is processed must not be responded to and all global atoms or data must be deleted. After receiving the return *wm_DDE_Terminate* message, the sending window can be closed.

Since *wm_DDE_Terminate* messages could be sent under extreme conditions, such as a fatal error that closes an application, the return *wm_DDE_Terminate* might not be processed. To protect against this case, the sending application should wait until it receives a *wm_DDE_Terminate* or until a certain time period has elapsed. It is good practice for all DDE messages that use the *PostMessage* function to have this time-out protection. Also the *IsWindow* function can be called, before calling *PostMessage,* to test whether the window handle parameter of the respondent is valid. These safeguards are not part of the DDE protocol but they do help you recover from catastrophic failures of the client or server. It's better to be safe than sorry.

Methods of exchanging data

There are five ways that a client and server can communicate:

1. A single data item can be exchanged with *wm_DDE_Request* and *wm_DDE_Data* messages.
2. A client can be informed when a data item is changed with *wm_DDE_Advise* and *wm_DDE_Data* messages. You can then

use *wm_DDE_Request* and *wm_DDE_Data* to get the data. This is the mechanism behind the application feature commonly referred to as a *hot link*. *wm_DDE_UnAdvise* is used to end these continual updates.

3. A combination of mechanisms 1 and 2, where the continual *wm_DDE_Data* messages actually pass the data, so that a *wm_DDE_Request* is not necessary.
4. A *wm_DDE_Poke* message tells the server to change the value of a data item.
5. A *wm_DDE_Execute* command passes a command macro to the server application to be interpreted and executed by the server.

Requesting a single data item

Typically, the client window will request to receive data from the server window. You can request a single data item with the following code:

```
procedure TClientWindow.RequestData(DataFormat: Word; Item: PChar;
  ServerHWnd: HWnd);
var
  ItemGlobalAtom: Word;
  lParam: Longint;
begin
  ItemGlobalAtom := GlobalAddAtom(Item);
  lParam := DataFormat or (ItemGlobalAtom shl 16);
  if PostMessage(ServerHWnd, wm_DDE_Request, HWindow, lParam) = 0
then
    GlobalDeleteAtom(ItemGlobalAtom);
end;
```

The *DataFormat* parameter is one of the clipboard formats, such as *cf_Text* or *cf_DIF*. If the server can render the data in the requested format, it can respond with a *wm_DDE_Data* message:

```
...
const
  DDE_FAckReq =      $8000;
  DDE_FDeferUpd =    $4000;
  DDE_FRelease =     $2000;
  DDE_FRequested =   $1000;
type
  TDDEData = record
    bitOptions: Word;
    cfFormat: Word;
```

```
  end;
...
procedure TServerWindow.ACKData(Item: PChar; DataFormat: Word; Data:
  Pointer; DataSize: Word);
var
  DataRecord: TDDEData;
  lParam: Longint;
  ItemGlobalAtom: Word;
  DataGlobalHandle: Word;
  DataGlobalPtr: Pointer;
begin
  DataGlobalHandle := GlobalAlloc(gmem_Moveable or gmem_DDEShare,
    SizeOf(TDDEData) + DataSize);
  if (DataGlobalHandle <> 0) then
  begin
    DataGlobalPtr := GlobalLock(DataGlobalHandle);
    if (DataGlobalPtr = nil) then GlobalFree(DataGlobalHandle)
    else
    begin
      DataRecord.cfFormat := DataFormat;
      DataRecord.bitOptions := DDE_FRequested or DDE_FAckReq;
      Move(@DataRecord, DataGlobalPtr, SizeOf(DataRecord));
      Move(Data, DataGlobalPtr + SizeOf(DataRecord), DataSize);
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom := GlobalAddAtom(Item);
      lParam := DataGlobalHandle or (ItemGlobalAtom shl 16);
      if PostMessage(ClientHWnd, wm_DDE_Data, HWindow, lParam) = 0
        then
      begin
        GlobalFree(DataGlobalHandle);
        GlobalDeleteAtom(ItemGlobalAtom);
      end
    end
  end;
end;
```

There are a couple of items to note in this example. First, the global memory is allocated as *gmem_DDEShare*. This allows both the server and client to have access to the data referenced by the handle. Second, the data block record is written in front of the actual data in global memory. This provides a way to specify the format of the data and a few bit field options which tell the receiving client how to respond. The two options specified are *DDE_FRequested*, which specifies that the *wm_DDE_Data* message is being sent in response to a *wm_DDE_FRequested* message, and *DDE_FAckReq*, which tells the receiving client to send a *wm_DDE_Ack* acknowledging receipt of the *wm_DDE_Data*

message. If the data format is *cf_Text*, then each line of data in the passed data pointer must be delimited by a carriage-return line-feed character pair, including the last line.

If the server cannot process the *wm_DDE_FRequested* message, then it must send a negative acknowledgement back to the client. This would be necessary if the server is unable to render the data item in the requested format. The general rule of thumb is to request the most complex formats first and to try simpler formats until you receive a positive acknowledgement. The negative acknowledgement would look like this:

```
...
ItemGlobalAtom := GlobalAddAtom(Item);
lParam := ItemGlobalAtom shl 16;
if PostMessage(ClientHWnd, wm_DDE_Ack, HWindow, lParam) = 0 then
  GlobalDeleteAtom(ItemGlobalAtom);
...
```

If the client does receive the *wm_DDE_Data* message, it can process the data that it receives. It must also check the *TDDEData* record at the beginning of the data pointer and, if *bitOptions* contains *DDE_FAckReq*, then it must send a positive acknowledgement back:

```
...
if ((DataRecord.bitOptions or DDE_FAckReq) <> 0) then
begin
  ItemGlobalAtom := GlobalAddAtom(Item);
  lParam := DDE_FAckReq or (ItemGlobalAtom shl 16);
  if PostMessage(ServerHWnd, wm_DDE_Ack, HWindow, lParam) = 0 then
    GlobalDeleteAtom(ItemGlobalAtom);
end;
...
```

If the *DDE_FRelease* bit option is set, the client is responsible for freeing the memory handle:

```
...
GlobalUnlock(ItemGlobalHandle);
if ((DataRecord.bitOptions or DDE_FRelease) <> 0) then
  GlobalFree(ItemGlobalHandle);
end;
...
```

The *DDE_FAckReq* or the *DDE_FRelease* flag (or both) should be sent in the *wm_DDE_Data* message. The global memory is freed by either the server, when it receives an acknowledgement, or by

the client. If neither flag is set, the server will not know when to free the global data and the client will not free it.

Though it may seem somewhat more complex than it needs to be, this portion of the DDE protocol is designed to achieve three important goals:

1. To ensure that all allocated global memory objects are de-allocated.
2. To allocate all global memory objects in a form that will be available to both client and server.
3. To prevent deadlock situations where one application is waiting for a response from the other application.

Understanding how these three goals operate in each of the DDE messages will clear up many of the complexities of the protocol.

Ongoing data transfers

A client can establish an ongoing link—a hot link—with the server to send notice of changes to a data item. This link is established by posting a *wm_DDE_Advise* message:

```
procedure TClientWindow.Advise(Item: PChar; DataFormat: Word);
var
  DataRecord: TDDEData;
  lParam: Longint;
  ItemGlobalAtom: Word;
  DataGlobalHandle: Word;
  DataGlobalPtr: Pointer;
begin
  DataGlobalHandle := GlobalAlloc(gmem_Moveable or gmem_DDEShare,
    SizeOf(TDDEData));
  if (DataGlobalHandle <> 0) then
  begin
    DataGlobalPtr := GlobalLock(DataGlobalHandle);
    if (DataGlobalPtr = nil) then GlobalFree(DataGlobalHandle)
    else
    begin
      DataRecord.cfFormat := DataFormat;
      DataRecord.bitOptions := DDE_FDeferUpd or DDE_FAckReq;
      Move(@DataRecord, DataGlobalPtr, SizeOf(DataRecord));
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom := GlobalAddAtom(Item)
      lParam := DataGlobalHandle or (ItemGlobalAtom shl 16);
      if PostMessage(ClientHWnd, wm_DDE_Advise, HWindow, lParam) = 0
        then
```

```
      begin
        GlobalFree(DataGlobalHandle);
        GlobalDeleteAtom(ItemGlobalAtom);
      end
    end
  end;
end;
```

Notice that the *TDDEData bitOptions* field has *DDE_FDeferUpd* set. This means that subsequent *wm_DDE_Data* message sent from the server or the client will actually contain the data. The *wm_DDE_Data* sent is equivalent to the *ACKData* procedure above except that the *DDE_FRequested* flag is not set. If the *DDE_FDeferUpd* field is not set, subsequent *wm_DDE_Data* messages will be sent without the data and the client will have to send *wm_DDE_Request* to get the data. If the *DDE_FDeferUpd* flag is set, then the response might be:

```
...
ItemGlobalAtom := GlobalAddAtom(Item);
lParam := ItemGlobalAtom shl 16;
  if PostMessage(ClientHWnd, wm_DDE_Data, HWindow, lParam) = 0 then
    GlobalDeleteAtom(ItemGlobalAtom);
...
```

If the server can render the data in the requested format, it should send a positive *wm_DDE_Ack*. If it cannot service the request, it should send a *wm_DDE_Ack*. Upon receipt of the *wm_DDE_Ack* message, the client can attempt to establish the link with a different clipboard data format.

Requesting the server to change a data value

The *wm_DDE_Poke* message is used to request a server to change the value of a data item. It can be used like this:

```
procedure TClientWindow.Poke(Item: PChar; DataFormat: Word; Data:
  Pointer; DataSize: Word);
var
  DataRecord: TDDEData;
  lParam: Longint;
  ItemGlobalAtom: Word;
  DataGlobalHandle: Word;
  DataGlobalPtr: Pointer;
begin
  DataGlobalHandle := GlobalAlloc(gmem_Moveable or gmem_DDEShare,
    SizeOf(TDDEData) + DataSize);
  if (DataGlobalHandle <> 0) then
```

```
    begin
      DataGlobalPtr := GlobalLock(DataGlobalHandle);
      if (DataGlobalPtr = nil) then
        GlobalFree(DataGlobalHandle)
      else begin
        DataRecord.cfFormat := DataFormat;
        Move(@DataRecord, DataGlobalPtr, SizeOf(DataRecord));
        Move(Data, DataGlobalPtr + SizeOf(DataRecord), DataSize);
        GlobalUnlock(DataGlobalHandle);
        ItemGlobalAtom := GlobalAddAtom(Item)
        lParam := DataGlobalHandle or (ItemGlobalAtom shl 16);
        if PostMessage(ServerHWnd, wm_DDE_Poke, HWindow, lParam) = 0
          then
        begin
          GlobalFree(DataGlobalHandle);
          GlobalDeleteAtom(ItemGlobalAtom);
        end
      end
    end;
end;
```

The code for *TClientWindow.Poke* is similar to that for *TServerWindow.ACKData*. The difference is that *wm_DDE_Poke* is sent from client to server and *wm_DDE_Data* is always sent from server to client. The server must send a positive acknowledgement if it can handle the *wm_DDE_Poke* and a negative acknowledgement if it can't. The server, in processing the *wm_DDE_Poke* message, can get the item name like this:

```
...
var
  ItemName : array[0..128] of Char;
...
ItemGlobalAtom := lParam shr 16;
GlobalGetAtomName(ItemGlobalAtom, @ItemName, SizeOf(ItemName));
...
```

Executing macro commands in the server

The *wm_DDE_Execute* message allows a client to send a string to the server to be executed as a command macro. If the server is a word processing application, then the command string would be in the form of the word processor's macro language. As is usually the case, the server must send a positive or negative acknowledgement of the disposition of the *wm_DDE_Execute* message. Here is an example of its use:

```
procedure TClientWindow.Execute(Command: Pointer; CommandSize: Word);
var
  lParam: Longint;
  DataGlobalHandle: Word;
  DataGlobalPtr: Pointer;
begin
  DataGlobalHandle := GlobalAlloc(gmem_Moveable or gmem_DDEShare,
    DataSize);
  if (DataGlobalHandle <> 0) then
  begin
    DataGlobalPtr := GlobalLock(DataGlobalHandle);
    if (DataGlobalPtr = nil) then
      GlobalFree(DataGlobalHandle)
    else begin
      Move(Data, DataGlobalPtr, DataSize);
      GlobalUnlock(DataGlobalHandle);
      lParam := DataGlobalHandle shl 16;
      if PostMessage(ServerHWnd, wm_DDE_Execute, HWindow, lParam) = 0
        then
      begin
        GlobalFree(DataGlobalHandle);
      end
    end
  end;
end;
```

The command string sent must be null-terminated and should have the following syntax (where brackets are part of the command and braces indicate optional elements):

```
[macroCall]{[macroCall]}
```

The *macroCall* is:

```
macroName(parameter1 {, parameter2 ...})
```

For example, if our word processor macro language has the ability to go to a certain line number, *goto_line(lineNo)*, and select the current paragraph, *select_current_paragraph*, we would send the command string, "[goto_line(50)][select_current_paragraph]", to select the paragraph in which the 50th line is present. For applications that both support DDE as a server and have an extensive macro language, the list of possible interactions is endless.

System topic

All applications that support DDE should maintain a special topic called "System". The System topic can be used by clients that might be unfamiliar with your application. Through this mechanism, such clients may be able to make inferences about how DDE is implemented in your application. The following is a list of data items that the System topic should support. Data elements in a data item should be separated by tabs.

1. *SysItems* is a list of System topic items.
2. *Topics* is a list of all the currently supported topics. Since some applications define a different topic for each open file, this list might change constantly.
3. *Formats* is a list of the supported clipboard formats for rendering data.

C H A P T E R

17

All about GDI

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.

To provide applications with graphics functionality, Windows has a set of functions called the Graphics Device Interface, or GDI. The GDI can be thought of as the graphics engine that Windows application use to display and manipulate graphics. GDI functions give your application drawing capabilities that are 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. Device independence is accomplished through the use of device drivers that translate GDI function calls into commands meaningful to the output device being used.

Display contexts

Unlike traditional DOS-based graphics programs, Windows programs never directly write to screen pixels, but rather to a logical entity called a *display context*. A display context is a virtual surface with associated attributes, such as a pen, brush, font, background color, text color, and current position.

When you call GDI functions to draw on a display context, the device driver associated with that display context translates that drawing action into appropriate commands. These commands reproduce the drawing action as accurately as possible on the display device, regardless of the display's capabilities. The display might be a low-resolution monochrome screen or a four-million-color screen.

Think of a display context as the canvas of an oil painting, where the window is the entire painting, including the frame. Instead of painting directly on a framed oil painting, you paint on a canvas, and then mount the canvas in the frame. In the same way, you paint graphics on a window's display context. A display context holds a set of *drawing tools*, such as pens, brushes, and fonts.

A display context is a Windows-managed element, much like a window element, except that a display context has no corresponding ObjectWindows object.

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.

The process of obtaining, using, and releasing display contexts is described in detail in the "Displaying graphics in windows" section of this chapter.

What's in a display context?

Although you will normally not need to alter most of the attributes of a display context, it is important to know just what the display context contains. This section briefly describes the properties of display contexts, including bitmaps, colors,

mapping, clipping, and drawing tools. Some of these topics are covered in greater detail elsewhere in this chapter as well.

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*.

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 will 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 that 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.

Drawing tools

To perform most of the actual drawing, the display context uses three tools: a *pen*, a *brush*, and a *font* tool. 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). 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. 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.

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. The attributes of these tools dictate the appearance of graphics drawn with GDI functions, such as *LineTo*, *Rectangle*, and *TextOut*. Pens dictate the appearance of drawn lines; brushes dictate the appearance of filled shapes; and fonts dictate the appearance of drawn text.

To assign the attributes of a drawing tool, a Windows program selects a logical or stock tool into a display context. A logical tool is one your program creates by filling the fields of a particular record, a *TLogPen*, *TLogBrush*, or *TLogFont*. 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.

Stock tools

Stock tools are created with the GDI function *GetStockObject*. For example:

```
var
  TheBrush: HBrush
begin
  TheBrush := GetStockObject(LtGray_Brush);
  ...
end;
```

where *LtGray_Brush* is an integer constant defined in the ObjectWindows unit *WinTypes*. Here is a list of all available 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*
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;
```

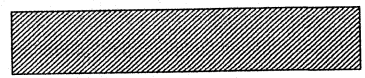
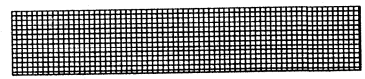
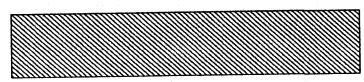
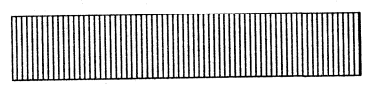
The *lbStyle* field holds an integer constant indicating the brush's style:

- *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.

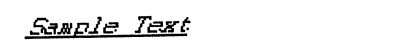
Table 17.3
Font pitch and family constants

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:


```
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;
```




```
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;
```

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.

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.

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.

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);
```

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 3, 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.

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 3, 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;
```

GDI drawing functions

This section describes the various API calls that you can use to draw things in windows.

Text drawing functions

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;
```

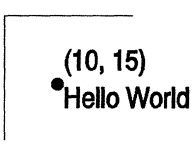


Figure 17.3
The results of the *TextOut* function

Line drawing functions

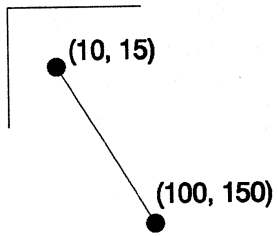
Line drawing functions use the specified display context's current pen to draw. Most lines are drawn using the *MoveTo* and *LineTo* functions. These functions affect a display context attribute called the *current position*. To use the analogy of drawing with a pencil and paper, the current position 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

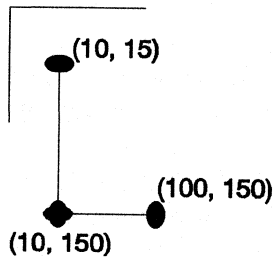


PolyLine

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



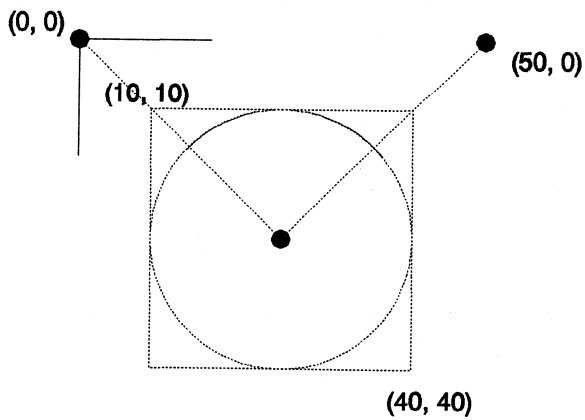
Arc

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



Shape drawing functions

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);
```

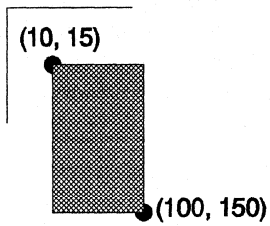


Figure 17.7
The results of the *Rectangle* function

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);
```

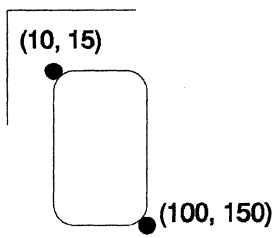


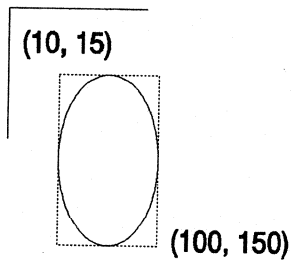
Figure 17.8
The results of the *RoundRect* function

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

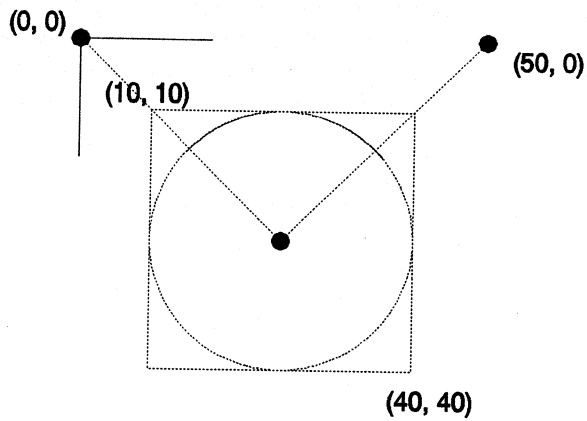


Pie and Chord

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);
```

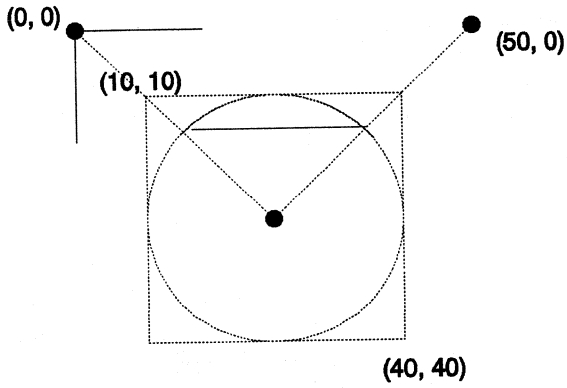
Figure 17.10
The results of the *Pie* function



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

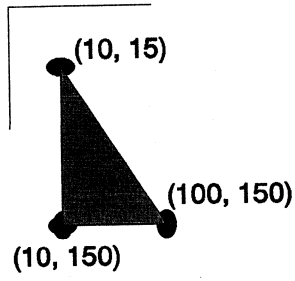


Polygon

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* we 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 following program 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 *Whitewater Resource Toolkit* for details on how to create or modify resources.

Creating resources

You can create resources using a resource editor or a resource compiler. Turbo Pascal for Windows 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 editor to copy resources from a .RES file into the program's already-compiled .EXE file. See the *Whitewater Resource Toolkit* 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.
- Use the Resource Compiler, as explained in this chapter.

Running the Resource Compiler

The use of resource editors is described fully in the *Whitewater Resource Toolkit*. This section provides a brief overview of how to use the Microsoft Resource Compiler to create resource files.

The Resource Compiler is a DOS application that takes a resource script file as input and generates a binary resource file. The Resource Compiler can also add the resources directly to the executable file, thereby eliminating the need to use Turbo Pascal's **$R** directive to add resources to a program. This is useful if you want to change the resources of an executable program when you don't have access to the Pascal source code.

The Resource Compiler is run at the DOS prompt or in a DOS window in Windows. The format is:

```
RC [options]  <.RC input file>   [.EXE output file]
```

where items in square brackets are optional.

For example:

```
RC -r SAMPPROG.RC
```

This creates a file, SAMPPROG.RES, from the input script file SAMPPROG.RC. The SAMPPROG.RES file must then be attached to the executable SAMPPROG.EXE file with the Turbo Pascal **$R** compiler directive, or by running the resource compiler on the .RES file:

```
RC SAMPPROG.RES
```

This attaches the existing binary resource file to the SAMPPROG.EXE file without requiring the use of the Turbo Pascal compiler.

Alternatively, you can combine both steps to directly modify an existing .EXE file to update the resources. In this case, use the command:

```
RC SAMPPROG.RC
```

This creates a SAMPPROG.RES file from the input script file SAMPPROG.RC and appends it directly to the SAMPPROG.EXE file, replacing any existing resources.

In either case, the Pascal program SAMPPROG.PAS must include commands for loading the resources into memory.

Valid command-line options for the Resource Compiler are:

Table 18.1
Resource Compiler command-line options

Option	Meaning
–r	Create a .RES file
–l	Create an application that uses LIM 3.2 EMS
–e	Create a driver which uses EMS memory
–m	Set multiple instance flag
–p	Create a private library
–t	Create a protected mode only application
–v	Verbose (print progress messages)
–d	Define a symbol
–fo	Rename .RES file
–fe	Rename .EXE file
–i	Add a path for INCLUDE searches
–x	Ignore INCLUDE environment variable
–k	Keep segments in .DEF file order (do not sort segments for fast load)

Resource Compiler script files

The .RC input script file is a text definition of the resources that will be compiled into binary format. Specifying resources using the Resource Compiler format is much like writing a program. Rather than provide an exhaustive definition of the Resource Compiler script language, a few examples should give you everything you need to know to compile and edit script files. Remember, in most cases it is easier to use a resource editor to create complex resources. You can also generate the script files directly from the resource editor if you like.

The sample below shows a typical resource script file, SAMPPROG.RC.

```
; SAMPPROG.RC --a sample resource script file
;             --this script illustrates the use of icons, strings,
;               menus, and dialog resources in Windows
;             --this script would be compiled with the resource
;               compiler and then appended to the pascal program
;               using the $R compiler directive

; include C-style header files for constant definitions:

#include <windows.h>
#include <wobjects.h>
#include <sampprog.h>

; define icons in separate files

SampleIcon      ICON    sample.ico

; strings can be defined numerically or symbolically

STRINGTABLE
BEGIN
  ids_NoProgrm, "Program unavailable"
  ids_Invalid, "Invalid entry"
  ids_Fatal, "Fatal error!"
;  .
;  . other strings go here
;  .
END

; Menus define commands for the user

SampleMenu MENU
BEGIN
  POPUP "&File"
    BEGIN
     MENUITEM "&New..\tCtrl+N",        cm_FileNew
     MENUITEM "&Open..\tCtrl+O",       cm_FileOpen
     MENUITEM "&Save..\tCtrk+S",       cm_FileSave
     MENUITEM SEPARATOR
     MENUITEM "&Print \tCtrl+P",       cm_FilePrint
    END
  POPUP "&Edit"
    BEGIN
    MENUITEM "&Undo \tAlt+BkSp",       cm_EditUndo
    MENUITEM SEPARATOR
    MENUITEM "&Cut \tShift+Del",       cm_EditCut
    MENUITEM "C&opy \tIns",            cm_EditCopy
    MENUITEM "&Paste \tShift+Ins",     cm_EditPaste
    MENUITEM "&Delete \tDel",          cm_EditDelete
    MENUITEM "C&lear All \tCtrl+Del", cm_EditClear
```

```
    END
  POPUP "&View"
    BEGIN
      MENUITEM "Summary \tF2",          cm_ViewSummary
      MENUITEM "Graph \tF3",            cm_ViewGraph
    END
END

; Accelerators provide "hot keys" to menus, commands

SampleAccelerators ACCELERATORS
BEGIN
  "^n", cm_FileNew
  "^o", cm_FileOpen
  "^s", cm_FileSave
  "^p", cm_FilePrint
  VK_DELETE, cm_EditCut, VIRTKEY, SHIFT
  VK_INSERT, cm_EditCopy, VIRTKEY
  VK_INSERT, cm_EditPaste, VIRTKEY, SHIFT
  VK_DELETE, cm_EditDelete, VIRTKEY
  VK_DELETE, cm_EditClear, VIRTKEY, CONTROL
  VK_BACK, cm_EditUndo, VIRTKEY, ALT
  VK_F2, cm_ViewSummary, VIRTKEY
  VK_F3, cm_ViewGraph, VIRTKEY
END

; Dialog boxes display information to the user

AboutBox  DIALOG 20, 20, 160, 80
CAPTION "About SAMPPROG"
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
FONT 8, "Helv"
BEGIN
  CTEXT "Sample Program"           -1,   0,  10, 160,  10
  CTEXT "Written in Turbo Pascal"  -1,   0,  26, 160,  10
  CTEXT "for Windows 3.0"          -1,   0,  36, 160,  10
  ICON  "Sample_Icon"              -1,   8,   8,   0,   0
  DEFPUSHBUTTON "OK",            IDOK,  60,  56,  40,  14
END

FileOpen DIALOG 20, 20, 204, 124
CAPTION "File Open"
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
FONT 8, "Helv"
BEGIN
  LTEXT    "File&name:",     -1,   6,   8,  36,  10
  EDITTEXT                  100,  42,   6,  98,  12, ES_AUTOHSCROLL
  LTEXT    "Directory:",     -1,   6,  20,  36,  10
  LTEXT    ""               101,  42,  20,  98,  10
  LTEXT    "&Files:",        -1,   6,  32,  64,  10
  LISTBOX  102, 6, 44, 64, 82, WS_TABSTOP | WS_VSCROLL | LBS_SORT
```

```
  LTEXT    "&Directories:", -1,  76,  32,  64,  10
  LISTBOX  103, 76, 44, 64, 82, WS_TABSTOP | WS_VSCROLL | LBS_SORT
  DEFPUSHBUTTON "OK",          IDOK, 146,   6,  50,  14
  PUSHBUTTON    "Cancel", IDCANCEL, 146,  24,  50,  14
END
```

The header file SAMPPROG.H consists of C-style constant definitions. A typical header file is shown below:

```
/* SAMPPROG.H    --header file of constants for SAMPPROG.RC  */

#define ids_NoProgrm        601
#define ids_Invalid         602
#define ids_Fatal           603

#define cm_FileNew          701
#define cm_FileSave         702
/*
  .
  . other constants included here
  .
*/
```

Resource Compiler hints

A few things to note about the Resource Compiler script programming language:

1. Comments begin with semi-colon (;) and run to the end of the line, or you can use C-style comments, text between /* and */.
2. Resource Compiler is *not* case sensitive with keywords such as BEGIN, END.
3. Resource Compiler *is* case sensitive to symbolic names, such as *cmFileNew*, *AboutBox*.
4. Resource Compiler is not sensitive to the use of whitespace.
5. You must use the **#include** directive to include header files of constants. These are defined using the C-style **#define** command.
6. You can use the **#rcinclude** directive to include other resource script files (*.RC or *.DLG); for example, to include compiled dialog boxes.
7. It is best to use symbolic constant names, rather than numbers, for constants such as string IDs, menu command IDs, and so on. These are normally defined using C-style constant definitions in a header file. You will also need to define Pascal

constants in your programs to refer to these constants symbolically.

8. Each menu item constant command ID and string ID must be unique.
9. Menus can be nested arbitrarily, thereby giving hierarchical menus.
10. Inside menu and dialog control items, the ampersand (&) tells the program to underline the next character. The underlined menus or dialog controls can be accessed by pressing the *Alt* key and the underlined letter.
11. Inside menu items, the text `\t` generates a tab character to line up text. This is often used to line up keyboard command equivalents.
12. Hot keys must be defined in an accelerator table. The text included in menus does not automatically create accelerators.
13. Controls such as static text which are never accessed under program control normally have the ID –1.
14. Dialog and control locations are defined in the order Left, Top, Width, Height.
15. Menu items, dialogs, and dialog control items, such as scrollbars, edit fields, and buttons, can be modified with Windows-style constants. Add them using the bitwise **or** operator, which is a vertical bar (|). For example, control items can have the styles WS_CHILD, WS_TABSTOP, and WS_BORDER.
16. Resource script files must be compiled using the Resource Compiler.
17. In most cases it is easier to use a resource editor rather than write complex resoure definitions using script files.
18. You must include code in your Turbo Pascal program to load and use the resources.

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
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(100));
  ...
end;
```

The code *PChar(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
  TWindow.Init(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 accelerator tables

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 will load the accelerator resource in the application object's *InitInstance* method:

```
procedure SampleApplication.InitInstance;
begin
  TApplication.InitInstance;
  HAccTable := LoadAccelerators(HInstance, 'SampleAccelerators');
end;
```

Often, you will 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 a command-based message 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, PChar(120)));
```

For more information on creating dialog objects, see Chapter 5, "Holding a dialog."

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 SampleWindow.GetWindowClass(var AWndClass: TWndClass);
begin
  TWindow.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 SampleWindow.GetWindowClass(var AWndClass: TWndClass);
begin
```

```
  TWindow.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 TestDialog.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 SampleWindow.Init(AParent: PWindowsObject; ATitle:
PChar);
var
  TextItem: array[0..255] of Char;
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(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, PChar(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, PChar(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, "All about GDI."

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.

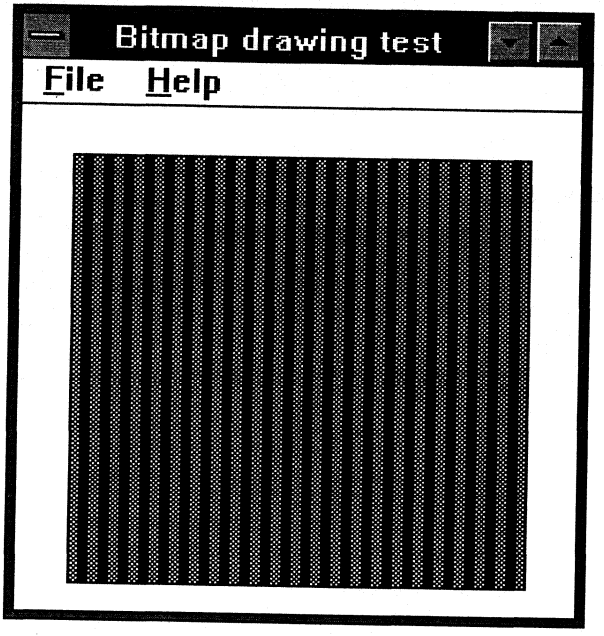
Using a bitmap to create a brush

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 SampleWindow.MakeBrush;
var
  MyLogBrush: TLogBrush;
begin
  HMyBit := LoadBitmap(HInstance, PChar(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
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(100));
  ALong.Lo := LoadBitmap(HInstance, PChar(503));
  ModifyMenu(Attr.Menu, 111, mf_ByCommand or mf_Bitmap, 211,
    PChar(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.

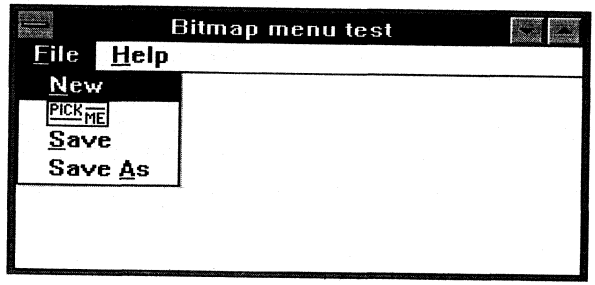


Figure 18.3
A menu that uses a bitmap as one of its selections

C H A P T E R

19

Standard application guidelines

One of the primary reasons for developing Windows-based software is its ease of learning and use. Windows users experience significant productivity gains primarily because operations common among programs are accessed in the same way. For example, the Edit menu has the same menu items and keyboard shortcuts in word processors, painting, and drawing applications. In other words, if you know how to use one Windows application, you know how to use them all.

To formalize the guidelines for common operation access, IBM has published a set of rules called Common User Access (CUA). While intended for OS/2 Presentation Manager applications, the CUA guidelines are just as appropriate for Windows applications, and are considered the industry standard. This section is based on IBM's published CUA guidelines.

Design principles

There are two primary goals of a CUA-compliant application: make users feel confident about the results of their actions, and let them control the flow of events.

Provide responses the user expects

The best way to make users feel confident is to ensure that the application responds in a consistent, dependable manner. Suggestions include:

- **Use metaphors.** If you model the application's behavior on something the users are familiar with, such as an existing paper-based system, they will have a better understanding of the application's underlying conceptual model. Plus, the users will feel comfortable sooner with the application.
- **Be consistent.** Implement consistency in your application's appearance and use. Use common commands (choosing menus, clicking icons, and so on) for common operations. Make screens for similar tasks look similar.
- **Avoid modes.** A mode is a state of the application where the users have restricted access to some features. It is often misused to force user activity in one particular direction. It is OK to use modes for short error messages, or during tool selection. For example, when the cursor is a pen, the user can draw but not drag around shapes. Be sure to provide visual acknowledgement of the mode, such as a different cursor or grayed menu items.

Allow the user to control the flow

The users will feel more confident and in control if they can dictate the path of activity. Allowing users to feel in control dispels some of the mystery surrounding computer applications. The program becomes an appliance.

- **List options visually.** Provide a visual list of available actions the user can take. Common methods include a palette of tools or colors and a menu of options.
- **Provide feedback.** Every user action should result in visual or audio feedback.
- **Encourage exploration.** Don't penalize users for exploring options. Often, this is the best way to learn a new application. Be sure to include an Undo option.

Standard appearance and behavior

Much of the Windows standard for application appearance and behavior is evident in the Program Manager and the supplied sample programs. For example, every application appears in a main window. Mouse clicks and double-clicks generally have the same effect from one application to the next.

General presentation

- **Use the object-action process.** A typical sequence of events for an application's operation is this: the user selects an item or object, and then performs an action on it. For example, in a word-processing application, the user might select text and then change its font or size. In a drawing program, the user might select a rectangle and then move it to a new location on the screen. While there are a number of ways to select objects, and even more ways to select actions, you should try to stick to this general model.
- **Choose your windows wisely.** A Windows application can appear in one or more windows. To design your application for optimum use, you should be aware of the different window types and their uses. The *main window* should unify the application, serving as its backbone. It actually represents the application in that minimizing it to an icon suspends the application's operation. The main window can also allow the user to choose a full-screen, or maximized, state. *Secondary windows*, such as popup windows and dialog boxes, should be spawned by the main window as a result of some main window action. Popup windows can represent an entirely separate module of the application, but should not be minimized.

Mouse and keyboard interaction

- **Show object selection.** When an object has been selected with a mouse click, provide a visual clue, such as a set of rectangles on its corners or a change in color or thickness. The selection mark should not permanently affect the object. After the user takes an action on the object, retain the selection. This allows the user to take additional actions. Deselect the object when the user selects something else.

- **Do not restrict the mouse pointer.** The mouse pointer should be free to move from one window or application to another. Do not limit its movement. Also, do not force its movement independent of the mouse. This will break the physical relationship between the mouse pointer and the mouse.
- **Use the keyboard selection cursor.** The keyboard selection cursor marks the spot future keyboard input will affect. For example, the blinking caret represents the keyboard selection cursor in a word-processing application. The keyboard selection cursor exists independently of the mouse pointer. The two are joined when mouse is clicked, however.
- **Mark the input focus.** The *input focus* is the point on the screen that is currently accepting input through the keyboard or mouse. For example, if the user selects a drop-down menu, that menu has the input focus. If the user highlights text, that text has the input focus. Always visually mark the object that has the input focus. For example, selected menu items are commonly displayed in reverse video and controls in a dialog are circumscribed with a dotted line. Often, when the user is dragging the mouse to select a group of items, the dragging area is circumscribed with an animated dotted line called a *marquee.*
- **Design a keyboard interface.** Use arrow keys, tabs, and keyboard accelerators to provide an alternative for users who don't have a mouse or care not to use it.
- **Conform to mouse click conventions.** Windows uses, but does not enforce, certain mouse clicking conventions. In general, use the left mouse button for selecting and manipulating objects, and the right button for other tasks, such as changing modes or bringing up a popup list of options. For the left mouse button, conform to the following conventions:

  Click selects an object.

  Double-click selects an object and causes some default or implied action.

  Click-drag manipulates the object's size and position or selects and highlights text.

  Ctrl-click adds the selected object to a set of previously selected objects.

  Shift-click extends a range of previously selected set of objects between the set's first object (its anchor) and the clicked object. (See the Windows File Manager for an example.)

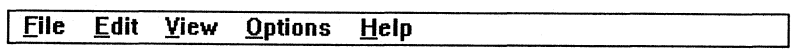
Menus

Windows already provides and enforces a fairly rigid standard for a pull-down menu's appearance. However, you do have some control over the styles you choose and the options you decide to implement. For example, you should use unique mnemonics for keyboard access to each top-level and drop-down menu item. You should use an ellipsis (...) following a menu item whose selection results in a dialog box. Use graying to mark a menu item as inactive.

Most applications, whether for graphics, business, education, science, or engineering, perform a particular set of operations for file manipulation, editing, and access to the help system. For this reason, CUA requires particular formats for File, Edit, and Help menus. In addition, it offers guidelines for View and Options menus. CUA requires that you put the top-level menu items in the following order, where Help is *always* last, regardless of how many other menus there are.

In order to produce help files that can be read by the Windows help system, you need to obtain the Help Compiler utility from Microsoft.

Figure 19.1
CUA standard menu bar



The following figures depict CUA-recommended menu items. Notice the recommended keyboard accelerators that appear next to some of the menu items.

Figure 19.2
CUA-recommended File menu items

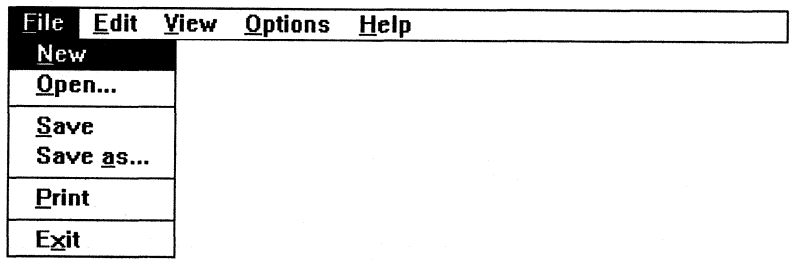


Figure 19.3
CUA-recommended Edit menu items

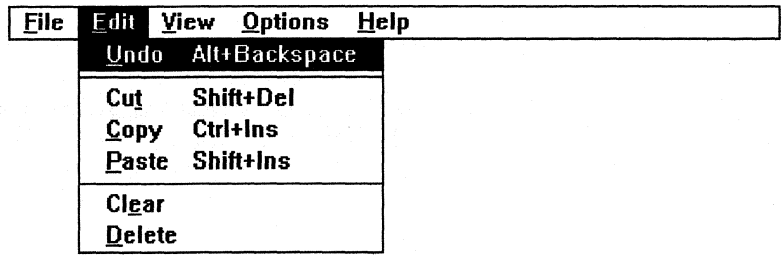


Figure 19.4
CUA-recommended View menu items

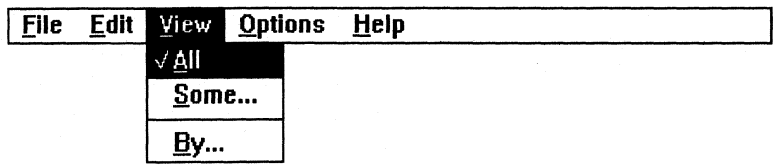


Figure 19.5
CUA-recommended Options menu items

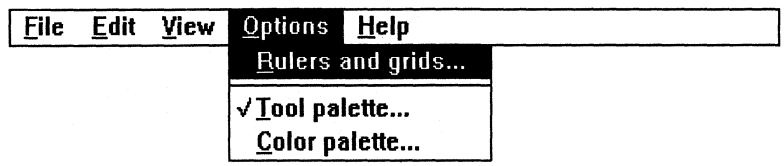
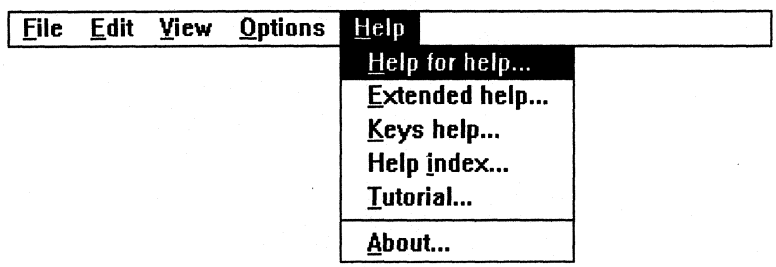


Figure 19.6
CUA-recommended Help menu items



Dialog boxes

There are so many ways to design dialog boxes that there are few guidelines. It is a good idea, however, to choose a style for your application and stick with it. There are two types of dialogs boxes, modal and modeless. Modal, which suspends operation of the program, should be used most often. Modeless, which allows regular operation of the program to continue, should be used only for specialized purposes and parallel activities, such as spell-checking a document.

CUA does require a particular style of dialog box for Open and Save As dialog boxes. ObjectWindows's stock file dialog box meets these requirements.

Design considerations

- **Choose a main window size.** Use only what you need to present the required information and present a usable client area.
- **Resize windows thoughtfully.** You have two choices for painting windows after they have been resized. You can paint the contents in its original proportion, allowing part to be clipped if the window is too small. This is best for what-you-see-is-what-you-get (WYSIWYG) applications. Or you can reformat the contents so they always fit well within the window's borders. This option might be more appropriate for a game or utility.
- **Use color sparingly.** Color should reinforce an established communication link with the user. It should be used consistently and the user should have the option to modify any color used.

Writing safe programs

Handling errors in an interactive user interface is much more complicated than in a command line utility. In a non-interactive application, it is quite acceptable (and indeed, expected) that errors cause the program to display an error message and terminate the program. In an interactive setting, however, the program needs to recover from errors and leave the user in an acceptable state. Don't allow errors to corrupt the information the user is working on or terminate the program, regardless of their nature. A program that meets these programming criteria can be considered "safe."

ObjectWindows facilitates writing safe programs. It promotes a style of programming that makes it easier to detect and recover from errors, especially the wily and elusive "Out of memory" error. It does this by promoting the concept of atomic operations.

All or nothing programming

An atomic operation is an operation that cannot be broken down into smaller operations. Or, more specific to our use, it is an operation that either completely fails or completely succeeds. Making operations atomic is especially helpful when dealing with memory allocation.

Typically, programs allocate memory in many small chunks. For example, when constructing a dialog box object, you allocate memory for the dialog box and for any other object that dialog owns, such as its child controls. Each of these allocations could potentially fail, and each possible failure requires a test to see if you should proceed with the next allocation or stop. If any allocation does fail, you need to deallocate any memory allocated successfully. Ideally, you would allocate everything and then check to see if any of your allocations failed. Enter the safety pool.

The safety pool

ObjectWindows sets aside a fixed amount of memory (8K) called the safety pool. If allocating memory on the heap dips into the safety pool, the system function *LowMemory* returns *True*. This indicates that further allocations are not safe and they may fail.

For the safety pool to be effective, the pool must be as large as the largest atomic allocation. In other words, it needs to be large enough to make sure that all allocations between checks of *LowMemory* will succeed; 8K should suffice in most applications.

When your application first comes up and creates its main window and its child windows, it already checks for the low memory condition. However, if your program creates windows or dialogs as dynamic objects any time after the main window is displayed, you must explicitly invoke the mechanism that checks for the low memory condition. These situations include creating dialog boxes or popup windows in response to a menu selection. MDI applications, however, already check for the low memory condition when creating child windows.

To check for low memory when creating windows or modeless dialogs, substitute your call to your window's *Create* method with a call to the *MakeWindow*, a *TApplication* method inherited by your application type. This method creates a window without checking for low memory:

```
procedure TMyWindow.Help(var Msg: TMessage);
```

```
var
  HelpWnd: PHelpWindow;
begin
  HelpWnd := New(PHelpWindow, Init(@Self, 'Help System'));
  HelpWnd^.Create;
end;
```

This method does check for low memory:

```
procedure TMyWindow.Help(var Msg: TMessage);
var
  HelpWnd: PHelpWindow;
begin
  HelpWnd := Application^.MakeWindow(New(PHelpWindow, Init(@Self,
    'Help System')));
  if HelpWnd <> nil then { window was successfully created }
end;
```

To specify your program's application object, use the global variable, *Application*, defined for every ObjectWindows application. *MakeWindow* returns a pointer to the window object, or **nil** if the creation was unsuccessful.

To check for low memory when executing modal dialogs, call your application type's inherited *ExecDialog* method instead of your dialog's *Execute* method.

```
procedure TTestWindow.RunDialog(var Msg: TMessage);
var
  TestDlg: PTestDialog;
  RetValue: Integer;
begin
  TestDlg := New(PTestDialog, Init(@Self, 'DIAL1'));
  RetValue := Application^.ExecDialog(TestDlg);
  if RetValue = id_Cancel then
    { dialog creation failed, or user cancelled }
end;
```

If successful, *ExecDialog* returns the dialog's return value, usually zero or a positive integer. If running the dialog fails, *ExecDialog* returns one of a few error constants, each negative integers. This is important to know because a dialog failure automatically results in the dialog object being disposed. Before disposing the dialog object, verify that its return value was not negative.

MakeWindow and *ExecDialog* notify the application object of a low memory condition by calling the application object's *Error* method with the constant *em_OutOfMemory* as an argument. The *Error*

method inherited from *TApplication* displays a message box alerting the user of the error condition.

Other window creation errors

There are other conditions, besides low memory, that can prevent a window or dialog box to be created and displayed properly. For example, Microsoft Windows might not permit the creation of an interface element due to low Windows memory or to incorrect values passed in a Windows function. An application must be informed of these errors, like low memory conditions, in order to recover from them.

These other window creation errors are automatically reported by the ObjectWindows methods that construct and create interface objects. *TWindowsObject* defines an *Integer* field, *Status*, whose value is zero if there is no error. If there has been an error, it may be set to one of the following values:

Table 19.1
Window creation error values

Error	Meaning
em_OutOfMemory	Memory allocation ate into the safety pool.
em_InvalidWindow	The window element could not be created.
em_InvalidClient	An MDI window object was not able to create its client window.
em_InvalidChild	One or more of the object's child windows generated an error.
em_InvalidMainWindow	The main window could not be created.

To check for errors in the window constructors and creation methods you write, set *Status* equal to one of these constants, or to one of your own constants. For example, if a window must open a file, but fails, it can report an error with a constant, such as *em_FileOpenError*. If *TWindow.Create* or *TDialog.Execute* return non-zero values when called by *MakeWindow* or *ExecDialog*, respectively, the calling method will automatically report the error to the *Error* method of your window type, which, by default, simply calls the *Error* method of your application type. You can override your window or application type's *Error* method to properly handle each error.

```
constructor EditorWindow.Init(AParent: PWindowsObject; ATitle: PChar;
  AFileName: PChar);
begin
  TWindow.Init(AParent; ATitle);
  if (Status >= 0) and not OpenTheFile(AFileName) then
    Status := em_FileOpenError;
end;
```

Major consumers

You should also check for low memory conditions for major consumers, which are window objects that allocate memory greater than the size of the safety pool, such as reading the entire contents of a file into memory. Major consumers should check *LowMemory* themselves, instead of waiting until they have finished all construction and then allowing *MakeWindow* to do so for them.

If a major consumer runs out of memory in the middle of constructing itself, it should set the *Status* flag to an error value and stop trying to allocate more memory. The *Status* flag will be checked in the *MakeWindow* method, which will report the error and dispose the window object.

Obviously, this is not quite as nice as being able to assume that your constructors work, but it is the only way to manage the construction of windows that exceed the size of your safety pool.

PART 4

Collections and streams

C H A P T E R

20

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 Turbo 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 Turbo 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 will be 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 will be discussed later.

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 10 clients are inserted into *ClientList*, its size will increase in increments of 5 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. Often you'll 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'll 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. ObjectWindows 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 will 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 is 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 de-allocating the data when you're done. And that's exactly what the call to *Dispose* does; it disposes each element in the collection, and then disposes 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 collecton 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'll want the objects to have something in common. In fact, you'll want them to have an abstract ancestor object in common.

As an example, here's a program that puts 3 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 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'll 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 a 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

21

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 a ObjectWindows 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.

Turbo 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. Turbo 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 VMT 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 VMT 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 a 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 the *Put* procedure. 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 group 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* will retrieve complex objects. Thus, if the object you retrieve from a stream is a window that owns child windows, the child windows will be 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'll 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
  TWindow.Load(S);                          { load the ancestor type }
  S.Read(Surreal, SizeOf(Surreal));         { read additional fields }
end;

procedure TMagritte.Store(var S: Stream);
begin
  TWindow.Store(S);                         { store the ancestor type }
  S.Write(Surreal, SizeOf(Surreal));        { write additional fields }
end;
```

Warning! 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 will 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 global 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 will need 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 read instances back from streams.

Registering standard objects

ObjectWindows defines stream registration records for all its standard objects. In addition, the *WObjects* unit defines a *RegisterWObjects* procedure that automatically registers all of the objects in that unit.

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: A complete example

In Chapter 20, "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* doesn'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 ObjectWindows 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* beause 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 will be defined and the registration records will be 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 will put 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 will be 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: 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'll 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 will be 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 group'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 make allowances for this, reading back the object from a stream will result 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 group's child window list. This way, when you *Load* the group 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 Turbo 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 6, "Global 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.

INDEX

C

D

E

F

G

H

K

L

M

N

O

P

R

S

T

U

V

W

X

Y