Microsoft® Windows

Software Development Kit

Programmer's Learning Guide

Version 2.0

Microsoft Corporation

Document No. 050051052-200-I02-1087
Part No. 00477

Contents

Figures

Chapter 1
Introduction

1.1 About This Guide

This guide is intended to help the experienced C programmer make the transition to writing applications that use the Microsoft® Windows 2.0 application program interface. The guide provides detailed explanations of how to use Windows functions, messages, and data structures to carry out useful tasks common to all Windows applications, and illustrates these explanations with sample applications that you can compile and run with Windows 2.0.

1.2 What You Need to Start

To start using this learning guide, you need the following:

- Experience using the Windows user interface and an understanding of the Windows user interface
- An understanding of the Windows style guidelines
- Experience writing C-language programs and using the standard C run-time functions

Before starting any development, you should install Windows 2.0 on your computer and learn how to use it. Be sure to learn the names, purpose, and operation of the various parts of a Windows screen, such as windows, dialog boxes, menus, controls, and scroll bars. Your own applications will rely heavily on these features, so it is very important for you to understand them so that you use them properly.

One goal of Microsoft Windows is to provide a common user interface for all applications. This ultimately helps the user by reducing the effort required to learn the user interface of a Windows application and helps you by clarifying the choices you have to make when designing a user interface. To achieve this goal, however, you must base your application's user interface design on the recommended application style guidelines described in the *Microsoft Windows Application Style Guide*. You should read this style guide before starting your design effort.

The C programming language is the preferred development language for Windows applications. Many of the programming features of Windows were designed with the C programmer in mind. Windows applications can also be developed in Pascal and assembly language, but these languages present additional challenges that you typically bypass when writing applications in the C language.

1.3 What Tools Do You Need?

To build most Windows 2.0 applications, you need the following tools:

- Microsoft C Optimizing Compiler: **cl**
- Microsoft Resource Compiler: **rc**
- Microsoft Segmented-Executable Linker: **link4**
- Microsoft Windows Icon Editor: IconEdit
- Microsoft Windows Dialog Editor: Dialog
- Microsoft Program Maintenance Utility: **make**
- Microsoft Symbolic Debug Utility: **symdeb**

To build Windows libraries and font resource files, you need the following additional tools:

- Microsoft Macro Assembler: **masm**
- Microsoft Windows Font Editor: FontEdit

Most of these tools are provided in the Microsoft Windows 2.0 Software Development Kit. The C Compiler and Macro Assembler are not. All are described more fully in *Microsoft Windows Programming Tools.*

1.4 Sample Applications

The sample applications in this guide are written in the C programming language and conform to the user-interface style recommended by Microsoft for Windows applications.

The source files for all sample applications are provided on the Learning Guide Samples disk supplied with the Microsoft Windows 2.0 Software Development Kit. It is recommended that you review the sample application sources while reading the corresponding description in this guide. For your convenience, the subdirectories containing the sample sources are named by chapter. You may also use the sources as a basis for your own applications.

1.5 What's in the Learning Guide

The following list briefly describes the contents of this learning guide, chapter by chapter:

- Chapter 1, "Introduction," serves as an introduction to the content and purpose of the *Microsoft Windows Programmer's Learning Guide.*
- Chapter 2, "Windows Overview," compares Windows to the standard C environment, provides a brief overview of Windows, and describes the Windows programming model and the Windows application-development process.
- Chapter 3, "A Generic Windows Application," shows how to create a simple Windows application called Generic. You'll then use this application as a basis for subsequent examples in this learning guide.
- Chapter 4, "Output to the Screen," introduces the graphics device interface (GDI) and shows how to use GDI tools to create your own output.
- Chapter 5, "Keyboard and Mouse Input," shows how to process input from the mouse and keyboard.
- Chapter 6, "Icons," shows how to create and display icons for your applications.
- Chapter 7, "The Cursor, the Mouse, and the Keyboard," explains the purpose of the cursor, the mouse, and the keyboard, and shows how to use them in your applications.
- Chapter 8, "Menus," shows how to create menus for your applications and how to process input from menus.
- Chapter 9, "Bitmaps," shows how to create and display bitmaps.
- Chapter 10, "Dialog Boxes and Controls," explains dialog boxes and controls, how to create dialog boxes, and how to fill them with controls. This chapter also shows how to use a control in a window other than a dialog box.
- Chapter 11, "File Input and Output," explains the **OpenFile** function, as well as rules about disk files.
- Chapter 12, "Printing," shows how to use a printer with Windows.
- Chapter 13, "The Clipboard," explains the clipboard and shows how to use it in your applications.
- Appendix A, "Fonts," shows how to create and load fonts, and how to use them in the **TextOut** function.

- Appendix B, "Memory Management," shows how to allocate global and local memory.
- Appendix C, "Windows Libraries," explains how to create a Windows library.

Microsoft Windows 2.0 runs with the MS-DOS® operating system, also known as DOS or PC-DOS. In this learning guide, MS-DOS will be referred to as DOS.

1.6 Notational Conventions

Here are a few notes about the typographic conventions used in this manual:

Convention	Usage
italic	Used for filenames, such as *generic.def*.

monospace	Used for code excerpts taken from the various files that make up the sample applications. For example:
	`if (hPrevInstance)`
	`    return (NULL);`

bold	Used for names of programs, fields, data types, structures, statements, options, registers, and keywords. These items appear exactly as they would in code.

Windows Functions

The Windows functions appear in bold; for example, **ShowWindow**, **GetDoubleClickTime**, or **BeginPaint**. In addition to these functions, the sample applications in this guide use several "locally defined" functions: functions that are defined in the *.h* files for their respective applications. To help you distinguish these locally defined functions from Windows functions, the locally defined functions appear in roman type, not bold; for example, AddExt, GenericWndProc, or UpdateListBox.

Local and Global Variables

These variables, like locally defined functions, appear in roman type; for example, lParam, hInstance, or OrgX. Since many of these variables have the same names as fields (hInstance, for example), it is necessary to make the typographic distinction between fields, in bold, and variables, in roman.

Chapter 2
Windows Overview

2.1 Overview

Microsoft Windows 2.0 has many features that the standard C environment does not. For this reason, Windows applications may, at first, seem more complex than standard C programs, which is understandable when you consider some of the additions that Windows offers:

- A graphical user interface featuring windows, menus, dialog boxes, and controls for applications
- Queued input
- Device-independent graphics
- Multitasking
- Data interchange between applications

This chapter describes these features and explains the impact they have on the way you develop and write applications. This chapter also provides a brief definition of Windows and explains the Windows programming model. It starts with a comparison of Windows and the standard C environment, and ends with a discussion of the Windows application-development process.

2.2 A Comparison: Windows and C

Most C programmers use the standard C run-time library to carry out a program's input, output, memory management, and other activities. The C run-time library assumes a standard operating environment consisting of a character-based terminal for user input and output, and exclusive access to system memory as well as the input and output devices of the computer. In Windows, these assumptions are no longer valid. Windows applications share the computer's resources, including the CPU, with other applications and interact with the user through a graphics-based display, a keyboard, and a mouse. The following sections describe some of the major differences between the standard C environment and Windows.

2.2.1 The User Interface

One of the principle design goals of Windows is to provide visual access to most, if not all, applications at the same time. In a multitasking environment, it is important to give all applications some portion of the screen through which they can interact with the user. In some systems, this access is granted by giving a selected program full use of the screen while other programs wait in the background. In Windows, every application has access to some portion of the screen at all times.

An application shares the display with other applications by using a "window" for interaction with the user. Technically, a window is little more than a rectangular portion of the system display that the system grants use of to an application. In reality, a window is a combination of useful visual devices, such as menus, controls, and scroll bars, that the user uses to direct the actions of the application.

In the standard C environment, the system automatically prepares the system display for your application, typically passing a file handle to the application that you can use to send output to the system display by using conventional C run-time or DOS system calls. In Windows, you must create your own window before carrying out any output or receiving any input. But once you create a window, Windows provides a host of information about what the user is doing with the window and automatically carries out many of the tasks the user requests, such as moving and sizing a window.

Another advantage to Windows is that although a standard C program has access to a single screen, a Windows application can create and use any number of windows to display information in any number of ways. There are some terminate-but-stay-resident C programs that take control of a portion of the display when another program is running, but these programs are entirely responsible for managing the screen. Such programs must also make sure that their use of the screen does not interfere with other programs and that the screen is properly restored to its original content when the program returns control to the program previously having control. In Windows, the screen is managed for you by Windows, which controls the placement and display of windows and ensures that no two applications attempt to access the same part of the system display at the same time.

2.2.2 Queued Input

One of the biggest differences between Windows and standard C programming is the method of input. In the C environment, a program reads from the keyboard by making an explicit call to a function, such as **getchar**. The function typically waits until the user presses a key before returning the character code to the program. In Windows, an application does not make explicit calls to read from the keyboard. Instead, Windows receives all input from the keyboard, mouse, and timer in its system queue, and automatically redirects the input to the application by copying it from the system queue to the application's queue. When the application is ready to retrieve input, it reads from its queue and dispatches the message to the appropriate window.

In the standard C environment, input is typically in the form of 8-bit characters from the keyboard. The standard input functions, **getchar** and **fscanf**, read characters from the keyboard and return ASCII or other codes corresponding to the keys pressed. A program may also intercept

interrupts from input devices such as the mouse and timer to use information from those devices as input. So in the standard C environment, input is either not very informative or is technically difficult to manage.

In Windows, input from the keyboard and mouse is provided automatically to every window created. Windows provides input in a uniform format, called an "input message," containing information about the input that far exceeds the type of information available in other environments. An input message specifies the system time, the position of the mouse, the state of the keyboard, the scan code of the key (if a key is pressed), the mouse button pressed, as well as the device generating the message. For example, there are two keyboard messages: WM_KEYDOWN and WM_KEYUP. These messages correspond to the press and release of a specific key. With each message, Windows provides a device-independent virtual keycode that identifies the key, no matter which keyboard it is on, the device-dependent scan code generated by the keyboard, as well as the status of other keys on the keyboard, such as the SHIFT, CONTROL, and NUMLOCK keys. Keyboard, mouse, and timer messages all have the same format and are all processed in the same manner.

2.2.3 Device-Independent Graphics

In Windows, you have access to a rich set of device-independent graphics operations. This means your application can draw lines, rectangles, circles, and complex regions simply, and can use the same calls and data to draw on a high-resolution graphics display as well as on a dot-matrix printer.

Windows requires device drivers to convert graphics output requests to output for a printer, plotter, display, or other output device. A device driver is a special executable library that an application can load and connect to a specific output device and port. Your application can then carry out graphics operations in the "context" of the specific device. A "device context" comprises the device driver, the output device, and the communications port.

2.2.4 Multitasking

Windows is a multitasking system. This means that more than one application can run at a time. In the standard C environment, there are no particular provisions for multitasking. C programs typically "assume" that they have exclusive control of all resources in the computer, including the input and output devices, memory, the system display, and even the CPU itself. In Windows, however, applications must share these valuable resources with all other applications that are currently running. For this reason, Windows carefully controls these resources and requires Windows applications to use a specific program interface that guarantees control.

For example, in the standard C environment, a program has access to all of memory that has not been taken up by the system, by the program, or by terminate-but-stay-resident programs. This means programs are free to use all of available memory for whatever they like and may access memory by whatever method they like.

In Windows, memory is a shared resource. Since more than one application can be running at the same time, you must cooperatively share memory to avoid exhausting the resource. Applications may allocate what they need from system memory; however, to make the most efficient use of memory, the Windows memory manager often moves or even discards memory blocks that have not been locked. This means you cannot assume that objects to which you have assigned a memory location remain where you put them. If there are several applications running, Windows may move and discard memory blocks often. Also, in Windows, there are two sources of allocated memory: global memory, for large allocations, and local memory, for small allocations.

Another example of a shared resource is the system display. In the standard C environment, the system typically grants your application exclusive use of the system display. This means you can use the display in any manner you like, from changing the color of text and background, to changing the video mode from text to graphics. Your applications can even directly access video memory to change the content of the screen. In Windows, your application must share the system display with other applications, so it does not, and must not, take control of the display.

2.3 The Windows Programming Model

The Windows programming model describes what elements an application has available to interact with the user. These elements are primarily windows, menus, dialog boxes, and the message loop. The following sections describe these elements in detail.

2.3.1 Windows

A window is the primary input and output device of any application. It is an application's only access to the system display. Since nearly all programs interact with the user in some way through the system display, Windows applications must use windows if they want to communicate with the user.

A window is a combination of a title bar, a menu bar, scroll bars, borders, or other features that occupy a rectangle on the system display. You list the features you want for a window when you create the window. Windows

then draws the window and otherwise manages it. Figure 2.1 shows the main features of a window:

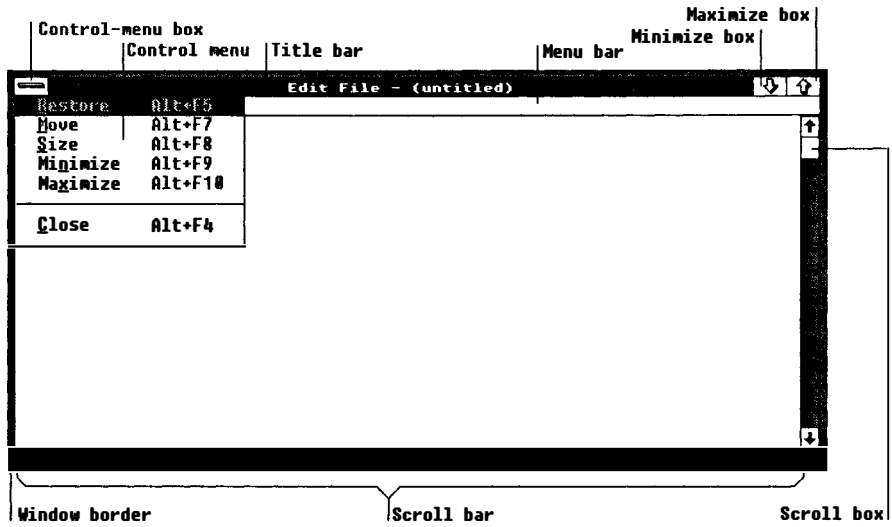


Figure 2.1 Window Features

Although an application creates a window and technically "owns" it, the management of the window is actually a collaborative effort between the application and Windows. Windows maintains the position and appearance of the window, manages the standard window features such as the border, scroll bars, and title, and carries out many tasks initiated by the user that directly affect the window. The application maintains everything else about the window—in particular, the client area in which an application is free to display anything it wants.

To manage this collaborative effort, Windows advises each window of changes that might affect it. So every window must have a corresponding "window function." A window function receives window-management messages that it must respond to appropriately. Window-management messages either specify actions for the function to take or are requests for information from the function.

2.3.2 Menus

Menus are the principle means of user input in a Windows application. A menu is a list of commands that the user can view and choose from. When you create an application, you supply the menu and command names.

Windows displays and manages the menus for you, sending a message to the window function when the user makes a choice. The message is your signal to carry out the command.

2.3.3 Dialog Boxes

A dialog box is a temporary window that you can create to give the user an opportunity to supply more information for a command. A dialog box contains one or more controls. A control is a small window that has a very simple input or output function. For example, an edit control is a simple window that lets the user enter and edit text. The controls in a dialog box give the user a method of supplying filenames, choosing options, and otherwise directing the action of the command.

2.3.4 The Message Loop

Since your application receives input through an application queue, the chief feature of any Windows application is the message loop. The message loop retrieves input messages from the application queue and dispatches them to the appropriate windows.

As shown in Figure 2.2, Windows collects hardware input, in the form of messages, in its system queue. It then copies this input to the appropriate application queue. The message loop in the application retrieves a message and dispatches it, through Windows, to the appropriate window function:

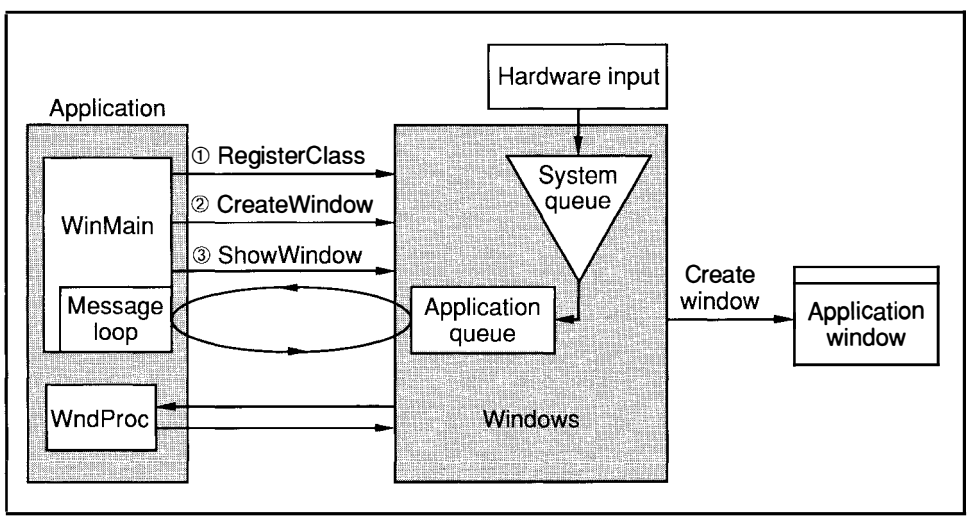


Figure 2.2 Processing Hardware Input

The window function can respond to an input message by calling Windows functions to carry out work on the window.

Figure 2.3 shows how Windows and an application collaborate to process keyboard input messages. Window receives keyboard input when the user presses and releases a key. Windows copies the keyboard messages from the system queue to the application queue. The message loop retrieves the keyboard messages, translates them into an ANSI character message, WM_CHAR, and dispatches the WM_CHAR message, as well as the keyboard messages, to the appropriate window function:

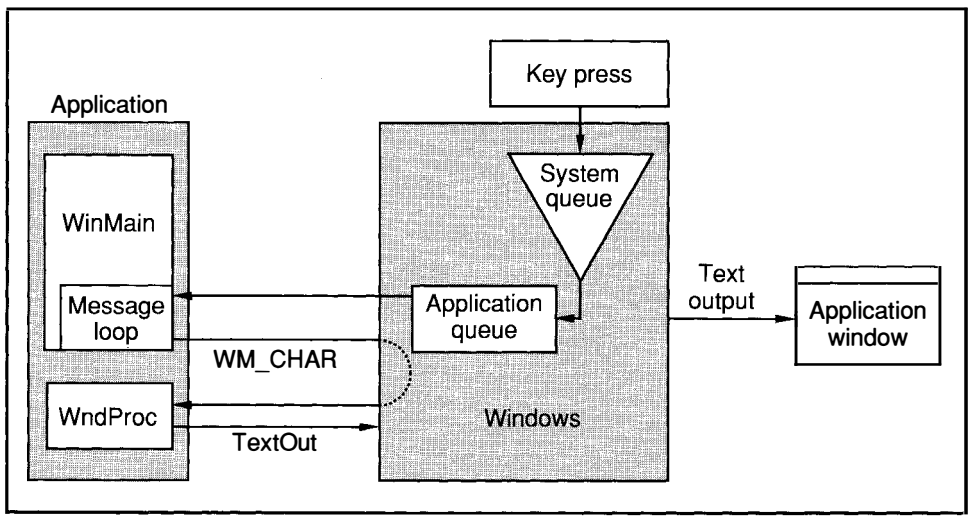


Figure 2.3 Processing Keyboard Input

The window function uses the **TextOut** function to display the character in the client area of the window.

Figure 2.4 shows how Windows sends window-management messages directly to a window function. After Windows carries out a request to destroy a window, it sends a WM_DESTROY message directly to the window function, bypassing the application queue. The window function must then signal the main function that the window is destroyed and the application should terminate. It does this by copying a WM_QUIT message into the application queue by using the **PostQuitMessage** function:

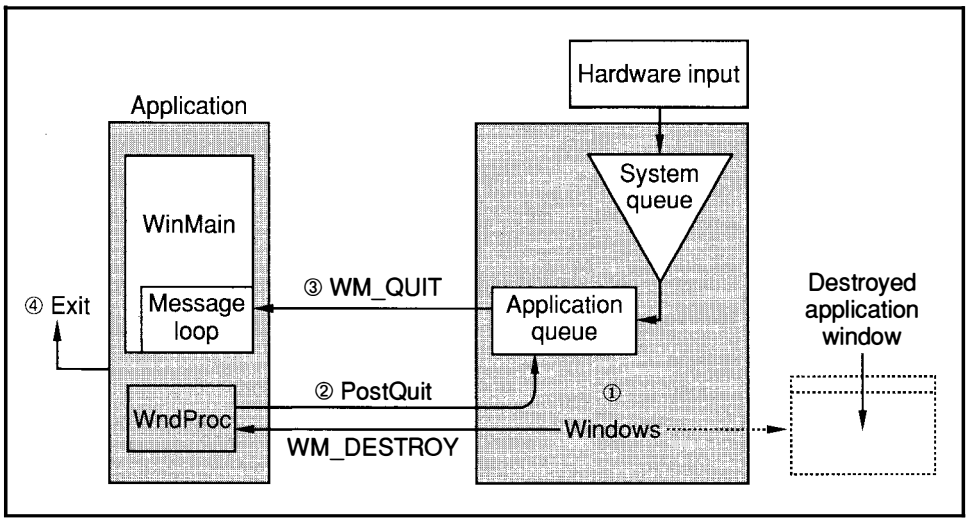


Figure 2.4 Processing Window-Management Messages

When the message loop retrieves the WM_QUIT message, the loop terminates and the main function exits.

2.4 The Windows Libraries

Windows functions, like C run-time functions, are defined in libraries. The Windows libraries, unlike the C run-time library, are special dynamic-link libraries that the system links your application with when it loads your application. Dynamic-link libraries are an important feature of Windows because they minimize the amount of code required by each application to run.

Windows consists of three main libraries, as described in the following list:

Library	Description
User	Provides window management. This library manages the overall Windows environment, as well as your windows.
Kernel	Provides system services, such as multitasking, memory management, and resource management.
GDI	Provides the graphics device interface.

2.5 Building a Windows Application

You build a Windows application by following these steps:

1. Write the **WinMain** and window functions and place them in C-language or assembly-language source files.
2. Write the menu, dialog box, and other resource descriptions and place them in a resource script file.
3. Use Icon Editor to create the cursors, icons, and bitmaps.
4. Use Dialog Editor to create dialog boxes.
5. Write the module definitions and place them in the module-definition file.
6. Compile and link all C-language sources.
7. Compile the resource script file and add it to the executable file.

Figure 2.5 shows the steps required to build a Windows application:

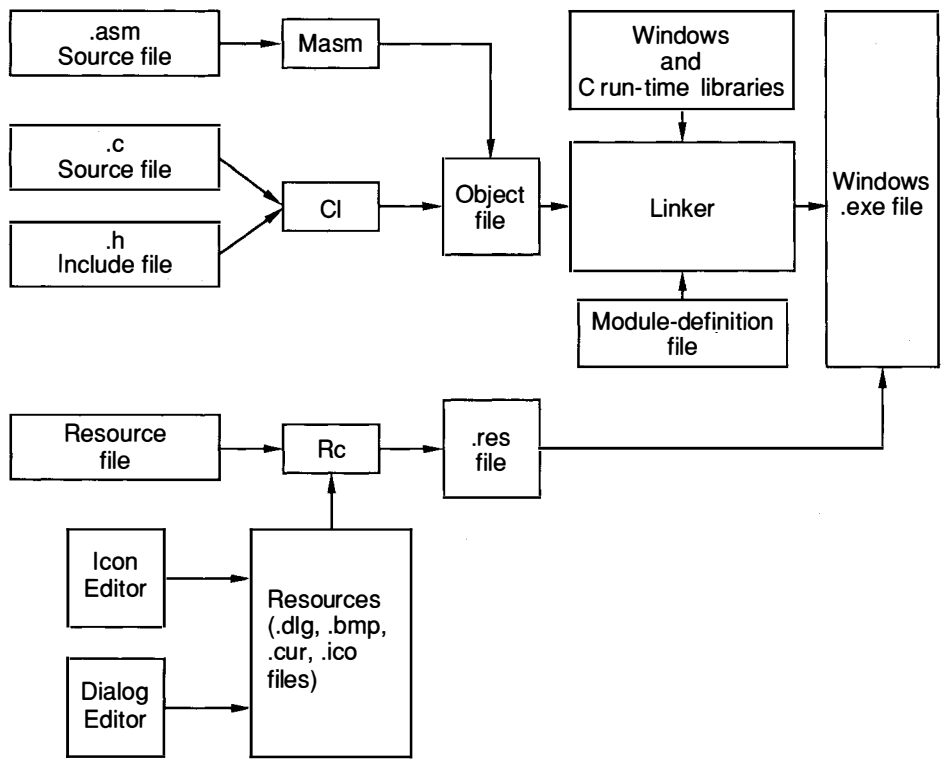


Figure 2.5 Building a Windows Application

To create a Windows application, you use many new development tools, as well as some familiar tools with new options. The following sections briefly describe each tool.

2.5.1 C Compiler

You may compile Windows applications by using the same options you use for standard C programs. However, Windows also requires two special options: **–Gw** and **–Zp**. The **–Gw** option adds the required Windows prolog and epilog code to each function. The **–Zp** option packs structures, ensuring that the structures used in your application are the same size as the corresponding structures used by Windows. The following shows a typical **cl** command:

```
cl -c -AS -Gw -Os -Zdp test.c
```

Since the object files generated by the C compiler must be linked with the Windows linker, **link4**, the **–c** option should be used to prevent **cl** from creating a file that is not executable with Windows.

2.5.2 Resource Compiler

Most Windows applications use a variety of resources, such as icons, cursors, menus, and dialog boxes. You must define these resources in a file called a "resource script file," then compile the file and add it to the application's executable file. When the application runs, it can load and use the resources from the executable file. The following is an example of a resource script file identifying two resources, a cursor and an icon:

```
bullseye CURSOR bullseye.cur
generic ICON generic.ico
```

To compile a resource script file and add it to an executable file, use the **rc** command. The following example shows a typical **rc** command:

```
rc generic.rc
```

The resource compiler is fully described in *Microsoft Windows Programming Tools.*

2.5.3 Linker

The Windows linker, **link4**, produces Windows-format executable files. The linker is similar to its previous versions except that it requires a module-definition file. This file names the callback functions in the application and defines the name of the application. The following is an example of a typical module-definition file:

```
NAME Generic

EXPORTS
    GenericWndProc
    AboutDlgFunc
```

To link a Windows application, you specify the name of the object files created by the compiler, the name of the Windows import library, the name of the module-definition file, and other options and files. The following example is a typical **link4** command:

```
link4 generic,/align:16,/map,slibw, generic.def
```

For more information on **link4** and the module-definition file, see *Microsoft Windows Programming Tools.*

2.5.4 Debugger

The symbolic debug utility, **symdeb**, helps you debug Windows applications. **Symdeb** lets you set breakpoints, view source-level code, and display symbolic information while debugging Windows applications. Although **symdeb** is a useful development tool, it is not described in this guide. If you want to read more about **symdeb**, see *Microsoft Windows Programming Tools.*

2.5.5 Resource Editors

There are three resource editors: Icon Editor, Font Editor, and Dialog Editor. These editors are Windows applications that allow you to create icons, cursors, and bitmaps (Icon Editor), fonts (Font Editor), and dialog-box descriptions (Dialog Editor). For more information on these editors, see *Microsoft Windows Programming Tools.*

2.5.6 Program Maintainer

The **make** program is a program maintainer that updates programs by keeping track of the dates of its source files. The **make** program is especially important because of the number of files required to create a Windows application. **Make** works with a **make** file that contains a list of the commands and files needed to build a Windows application. The commands compile and link the various files. **Make** executes the commands only if the files named in those commands have changed. This saves time if, for instance, you have made only a minor change to a single file. The following example shows the content of a typical **make** file for a Windows application:

```
generic.obj: generic.c generic.h
    cl -c -AS -Gw -Os -Zp generic.c

generic.exe: generic.obj generic.def generic.rc
    link4 generic,/align:16,/map, slibw, generic.def
    rc generic
```

Typically, **make** files have the same name as the applications they build, although any name is allowed. The following example shows a typical **make** command:

```
make generic
```

2.6 Tips for Writing Windows Applications

When writing Windows applications, remember the following general rules:

- Do not use C run-time console-input and -output functions, such as **getchar**, **putchar**, **scanf**, and **printf**.
- You may use the C run-time memory-management functions **malloc**, **calloc**, **realloc**, and **free**, but be aware that Windows translates these functions to its own local-heap functions, **LocalAlloc**, **LocalReAlloc**, and **LocalFree**. Since local-heap functions don't always operate exactly like C run-time memory-management functions, you may get unexpected results.
- Do not use C run-time file-input and -output functions to access serial and parallel ports. Instead, use the communications functions, which are described in detail in the *Microsoft Windows Programmer's Reference*.
- You can use the C run-time file-input and -output functions to access disk files. In particular, use the Windows **OpenFile** function and the low-level, C run-time input and output functions. Although the C run-time stream input and output functions can be used, you do not get the advantages of opening and managing files with **OpenFile**.
- Do not take exclusive control of the CPU—it is a shared resource. Although Windows is a multitasking system, it is non-preemptive. This means it cannot take control back from an application until the application releases it. A cooperative application carefully manages access to the CPU and gives other applications ample opportunity to execute.
- Do not attempt to directly access memory or hardware devices such as the keyboard, mouse, timer, display, and serial and parallel ports. Windows requires absolute control of these resources to ensure equal, uninterrupted access for all applications that are running.

Chapter 3
A Sample Application: Generic

3.1 Introduction

This chapter explains how to create a simple Microsoft Windows application called Generic. The Generic application demonstrates principles explained in Chapter 2, "Windows Overview," and illustrates the basic steps needed to develop a Windows application. Generic will also be used as basic code for all further sample applications in this guide.

3.2 The Generic Application

Generic is a standard Windows application; that is, it meets the recommendations for user-interface style given in the *Microsoft Windows Application Style Guide*. Generic has a main window, a border, a system menu (called a Control menu in the user manuals), and maximize and minimize boxes, but no other features. The system menu includes an About command, which, when chosen by the user, displays an About dialog box describing Generic. The completed Generic, with an About dialog box, looks like Figure 3.1 when displayed:

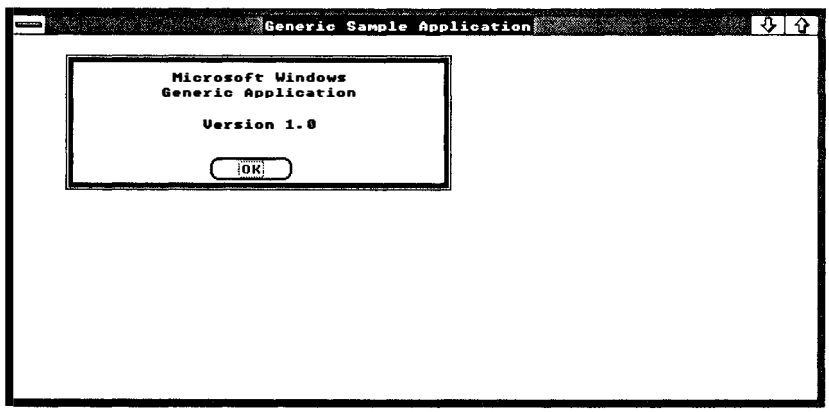


Figure 3.1 Generic with an About Dialog Box

Generic is important not for what it can do, but for what it provides: a template for writing Windows applications. Building it helps you understand how Windows applications are put together and how they work.

3.3 A Windows Application

A Windows application is any application that is specifically written to run with Windows and that uses the Windows application program interface (API) to carry out its tasks. A Windows application has the following basic components:

- A **WinMain** function
- A window function

The **WinMain** function is the entry point for the application and is similar to the main function used in the standard C environment.

A window function is something new. It is a callback function; that is, a function that you register with Windows to be "called back" when Windows needs to carry out work on a window. You never call a window function directly. Instead, you let Windows call the window function with requests to carry out specific tasks or to return information.

3.4 The WinMain Function

Every Windows application must have a **WinMain** function. An application cannot run without it. Much like the main function in standard C programs, **WinMain** is the entry point for the application. In most Windows applications, the **WinMain** function does the following:

- Registers the window classes to be used in the application and carries out other initializations.
- Creates a main window and possibly other windows to be used by the application.
- Starts a message loop to process messages from the application queue.
- Terminates the application when the message loop retrieves a WM_QUIT message.

The **WinMain** function has the following form:

```
int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;              /* current instance              */
HANDLE hPrevInstance;          /* previous instance             */
LPSTR lpCmdLine;               /* command line                  */
int nCmdShow;                  /* show-window type (open/icon) */
{
}
```

The **WinMain** function uses the **PASCAL** calling convention. Since Windows calls this function directly and always uses this convention, **PASCAL** is required. Windows passes **WinMain** four parameters: hInstance receives the instance handle of the application; hPrevInstance receives the handle of the previous instance of the application; lpCmdLine receives a long pointer to a null-terminated command line; and nCmdShow receives an integer to be passed to the **ShowWindow** function, which is used to display the application's main window. The nCmdShow parameter defines how the window should be displayed: as an open window or as an icon. For more information on handles, see Section 3.4.2, "Handles."

3.4.1 Windows Data Types

The **WinMain** function uses several nonstandard data types to define its parameters. For example, it uses the **HANDLE** data type to define the hInstance and hPrevInstance parameters, and the **LPSTR** data type to define the lpCmdLine parameter. In general, Windows uses many more data types than you would find in a typical C program. Although the Windows data types are often equivalent to familiar C data types, they are intended to be more descriptive and should help you better understand the purpose of a given variable or parameter in an application.

The Windows data types are defined in the *windows.h* include file. The Windows include file is an ordinary C-language source file that contains definitions for all the Windows special constants, variables, data structures, and functions. To use these definitions, you must include the *windows.h* file in each source file. Place the following line at the beginning of your source file:

```
#include "windows.h" /* Required for all windows applications */
```

The following is a list of some of the more common Windows data types:

Type	Meaning
WORD	Specifies a 16-bit, unsigned integer.
LONG	Specifies a 32-bit, signed integer.
HANDLE	Identifies a 16-bit, unsigned integer to be used as a handle.
HWND	Identifies a 16-bit, unsigned integer to be used as a handle to a window.
LPSTR	Specifies a 32-bit pointer to a **char** type.
FARPROC	Specifies a 32-bit pointer to a function.

The following is a list of some commonly used structures:

Structure	Description
MSG	Defines the fields of an input message.
WNDCLASS	Defines a window class.
PAINTSTRUCT	Defines a paint structure used to draw within a window.
RECT	Defines a rectangle.

3.4.2 Handles

The **WinMain** function has two parameters, hPrevInstance and hInstance, that are called handles. A handle is an unique integer that Windows uses to identify an object created or used by an application. Windows uses a wide variety of handles, identifying objects such as application instances, windows, menus, controls, allocated memory, output devices, files, GDI pens and brushes, and many more.

Most handles are index values for internal tables. Windows uses handle indexes to access the information stored in the table. Typically, your application has access only to the handle, and not to the data. When you need to examine or change the data, you supply the handle and Windows does the rest. This is one means Windows has of protecting data in its multitasking environment.

3.4.3 Managing Your Instances

Not only can you run more than one application at a time in Windows, you can also run more than one copy of the same application at a time. To distinguish one copy from another, Windows supplies a unique instance handle each time it calls the **WinMain** function to start the application. An instance is a separately executing copy of an application, and an instance handle is an integer that uniquely identifies an instance.

In some multitasking systems, if you run multiple copies of the same application, the system loads a fresh copy of the application's code and data into memory and executes it. In Windows, when you start a new instance of the application, only the data for the application is loaded. Windows uses the same code for all instances of the application. This is a way to save as much space as possible for other applications and for data. However, this method requires that the code segments of your application remain unchanged for the duration of the application. This means that you must not store data in a code segment or change the code while the program is running.

For most Windows applications, the first instance has a special role. Since many of the resources an application creates, such as window classes, are generally available to all applications, only the first instance of an application creates these resources. All subsequent instances may use the resource without creating them. To help you determine which is the first instance, Windows sets the hPrevInstance parameter of **WinMain** to NULL if there are no previous instances. The following example shows how to check for a previous instance:

```
int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;
HANDLE hPrevInstance;
LPSTR lpCmdLine;
int nCmdShow;
{
    if (!hPrevInstance)
      .
      .
      .
}
```

You can keep the user from starting more than one instance of an application by checking the hPrevInstance parameter when the application starts and returning to Windows immediately if the parameter is not NULL. The following example shows how to do this:

```
if (hPrevInstance)
    return (NULL);
```

3.4.4 Registering the Window Class

Before you can create any window you must have a window class. A window class is a template that defines the attributes of a window, such as the shape of the window's cursor and the name of the window's menu. Although Windows provides some predefined window classes, most applications define their own window classes in order to control every aspect of the way their windows operate.

You must register a window class before you can create a window that belongs to that class. You register a window class by filling a **WNDCLASS** structure with information about the class and passing it as a parameter to the **RegisterClass** function. Following is a list of the fields in the **WNDCLASS** structure:

Field	Description
lpszClassName	Points to the name of the window class. A window class name must be unique; that is, different applications must use different class names.

hInstance	Specifies the application instance that is registering the class.
lpfnWndProc	Points to the window function used to carry out work on the window.
style	Specifies the class styles, such as automatic redrawing of the window when moved or sized.
hbrBackground	Specifies the brush used to paint the window background.
hCursor	Specifies the cursor used in the window.
hIcon	Specifies the icon used to represent a minimized window.
lpszMenuName	Points to the resource name of a menu.
cbClsExtra	Specifies the number of extra bytes to allocate for this structure.
clWndExtra	Specifies the number of extra bytes to allocate for all the structures created with this class.

Some fields, such as **lpszClassName**, **hInstance**, and **lpfnWndProc**, must be assigned values. Other fields can be set to NULL to direct Windows to use a default attribute for windows created using the class. The following example shows how to register a window class named "Generic":

```
WNDCLASS WndClass;
long FAR PASCAL GenericWndProc(HWND, unsigned, WORD, LONG);
    .
    .
    .

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;
HANDLE hPrevInstance;
LPSTR lpCmdLine;
int nCmdShow;
{
    if (!hPrevInstance) {

        WndClass.lpszClassName = (LPSTR) "Generic";
        WndClass.hInstance = hInstance;
        WndClass.lpfnWndProc = GenericWndProc;
        WndClass.style = NULL;
        WndClass.hbrBackground = GetStockObject(WHITE_BRUSH);
        WndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
        WndClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
        WndClass.lpszMenuName = (LPSTR) NULL;
        WndClass.cbClsExtra = NULL;
        WndClass.clWndExtra = NULL;
```

```
        if (!RegisterClass(&WndClass))
            return (NULL);
            .
            .
            .
    }
}
```

In this example, the class name is "Generic". Notice that the name is cast using the **LPSTR** data type. This is to ensure that the **lpszClassName** field is assigned a 32-bit pointer to the string. The **hInstance** field receives the instance handle passed to the **WinMain** function. The **lpfnWndProc** field receives a pointer to the window function, GenericWndProc. This specifies that GenericWndProc will be the function that carries out tasks for the window.

To assign the address of the GenericWndProc function to the **lpfnWndProc** field, you must declare the function somewhere before the assignment statement. Most Windows applications use function prototypes for function declaration in order to take advantage of the Microsoft C Compiler's automatic type-checking and casting. The following is the correct prototype for GenericWndProc:

```
long FAR PASCAL GenericWndProc(HWND, unsigned, WORD, LONG);
```

The **hbrBackground** field receives a handle to a built-in white brush. The **GetStockObject** function returns handles to a variety of built-in drawing objects, such as pens, brushes, and fonts. The **hCursor** field receives a handle to a built-in cursor. The **LoadCursor** function returns handles to built-in or application-defined cursors. In this case, the NULL and IDC_ARROW arguments specify the built-in arrow cursor. The **hIcon** field receives a handle to a built-in icon. The **LoadIcon** function returns handles to built-in or application-defined icons. In this case, the NULL and IDI_APPLICATION arguments specify the built-in application icon.

The **style**, **lpszMenuName**, **cbClsExtra**, and **cbWndExtra** fields are set to NULL. This means Windows will supply default attributes when a window belonging to this class is created. Note that the **LPSTR** cast is required for the **lpszMenuName** assignment.

After you assign values to the **WNDCLASS** structure fields, you register the class by using the **RegisterClass** function. If registration is successful, the function returns TRUE. Otherwise, it returns FALSE. You should check the return value since if you cannot register a class, you cannot create your windows. If the registration fails, you should terminate the application.

Although the **RegisterClass** function requires a 32-bit pointer to a **WNDCLASS** structure, in the previous example, the address operator (&) generates only a 16-bit address. This is an example of an implicit cast carried out by the C compiler. The Windows include file contains

prototypes for all Windows functions. These prototypes specify the correct types for each function parameter, and the C compiler casts to these types automatically. In a few cases, you may need to provide an explicit cast or override a cast, but otherwise you can rely on the C compiler to cast appropriately.

3.4.5 Creating a Window

You can create a window by using the **CreateWindow** function. This function directs Windows to create a window that has the specified style and belongs to the specified class. **CreateWindow** takes several parameters: the name of the window class, the window title, the window's style, the window position, the parent window handle, the menu handle, the instance handle, and 32-bits of additional data. The following example creates a window belonging to the "Generic" class:

```
hWnd = CreateWindow("Generic",              /* window class     */
    "Generic Sample Application",           /* window name      */
    WS_OVERLAPPEDWINDOW,                    /* window style     */
    CW_USEDEFAULT,                          /* x position       */
    CW_USEDEFAULT,                          /* y position       */
    CW_USEDEFAULT,                          /* width            */
    CW_USEDEFAULT,                          /* height           */
    NULL,                                   /* parent handle    */
    NULL,                                   /* menu or child IO*/
    hInstance,                              /* instance         */
    NULL);                                  /* additional info */
```

This example creates an overlapped window that has the style WS_OVERLAPPEDWINDOW and that belongs to the Generic window class. The window caption is "Generic Sample Application".

Since the CW_USEDEFAULT value is specified for the position, width, and height parameters, Windows will place the window at a default position and give it a default width and height. The default position and dimensions depend on the system and on how many other applications have been started. Windows does not display the window until you call the **ShowWindow** function.

When you create a window, you may specify its parent (used with controls and child windows) and its menu. An overlapped window should not have a parent, so this parameter should be set to NULL. An overlapped window may have a menu, but in this case, NULL specifies that the class menu is desired. If there is no class menu, then the window will have no menu.

You must specify the instance of the application that is creating the window. Windows uses this instance to make sure that the window function supporting the window uses the data for this instance and not some other. The last parameter is for additional data to be used by the window function when the window is created. This window takes no additional data, so the parameter is set to NULL.

If **CreateWindow** successfully creates a window, it returns a handle to the new window. You can use the handle to carry out tasks on the window, such as showing it or updating its client area. If **CreateWindow** cannot create the window, it returns NULL. Whenever you create a window, you should check for a NULL handle and respond appropriately. For example, in the **WinMain** function, if you cannot create your application's main window, you should terminate the application; that is, return control to Windows.

3.4.6 Showing and Updating a Window

Although **CreateWindow** creates a window, it does not automatically display the window on the system display. Instead, it is up to you to display the window by using the **ShowWindow** function and to update the window's client area by using the **UpdateWindow** function.

The **ShowWindow** function directs Windows to display the new window, which has the handle hWnd. For the application's main window, **WinMain** should call **ShowWindow** soon after creating the window, and should pass the nCmdShow parameter to it. The nCmdShow parameter defines how the window is displayed: as an open window or as an icon. After calling **ShowWindow**, **WinMain** should call the **UpdateWindow** function. The following example illustrates how to show and update a window:

```
ShowWindow(hWnd, nCmdShow);          /* Shows the window       */
UpdateWindow(hWnd);                  /* Sends WM_PAINT message*/
```

3.4.7 Creating the Message Loop

Once the **WinMain** function has created and displayed a window, it can begin its primary duty: to read messages from the application queue and dispatch them to the appropriate window. **WinMain** does this by creating a message loop. A "message loop" is a program loop, typically created by using a **while** statement, in which **WinMain** retrieves messages and dispatches them.

Windows does not send input directly to an application. Instead, it places all mouse and keyboard input into an application queue (along with messages posted by Windows and other applications). The application must read the application queue, retrieve the messages, and dispatch them so the appropriate window function can process them. The simplest possible message loop consists of the **GetMessage** and **DispatchMessage** functions. This loop has the following form:

```
MSG msg;
    .
    .
    .

while (GetMessage(&msg, NULL, NULL, NULL)) {
    DispatchMessage(&msg);
}
```

In this example, the **GetMessage** function retrieves a message from the application queue and copies it into the structure, msg. The NULL arguments indicate that all messages should be processed. The **DispatchMessage** function directs Windows to send each message to the appropriate window function. Every message an application receives, except the WM_QUIT message, belongs to one of the windows created by the application. Since an application must not call a window function directly, the **DispatchMessage** function is required to make sure the messages get to the appropriate function.

Depending on what the application does, you may need a more complicated message loop. In particular, if you wish to process character input from the keyboard, you need to translate each message you receive by using the **TranslateMessage** function. Your message loop should then look like this:

```
while (GetMessage(&msg, NULL, NULL, NULL)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

The **TranslateMessage** function looks for matching WM_KEYDOWN and WM_KEYUP messages and generates a corresponding WM_CHAR message for the window that contains the ANSI character code for the given key.

A message loop may also contain functions to process menu accelerators and keystrokes within dialog boxes. Again, this depends on what your application actually does.

Windows places input messages in an application queue when the user moves the mouse cursor (pointer) in the window, or presses or releases a mouse button when the mouse cursor is in the window, or presses or releases a keyboard key when the window has the input focus. The window manager first collects all keyboard and mouse input in a system queue, then copies the corresponding messages to the appropriate application queue.

The message loop continues until **GetMessage** returns NULL, which it does only if it retrieves the WM_QUIT message, which is a signal to terminate the application. It is usually posted (placed in the application queue) by the window function of the application's main window.

3.4.8 Yielding Control

Windows is a non-preemptive multitasking system. This means that Windows cannot take control from an application. The application must yield control before Windows can reassign control to another application.

To make sure that all applications have equal access to the CPU, the **GetMessage** function automatically yields control when there are no messages in an application queue. This means that if there is no work for the application to do, Windows can give control to another application. Since all applications have a message loop, this implicit yielding of control guarantees sharing of control.

In general, you should rely on the **GetMessage** function to yield for your application. Although an explicit **Yield** function is available, you should avoid using it. Since there may be times when your application must keep control for a long time, such as when writing a large buffer to a disk file, you should try to minimize the work and provide a visual clue to the user that a lengthy operation is underway.

3.4.9 Terminating an Application

Your application terminates when the **WinMain** function returns control to Windows. You can return control at any time before starting the message loop. Typically, an application checks each step leading up to the message loop to make sure each window class is registered and each window is created. If there is an error, the application can display a message before terminating.

Once the **WinMain** function enters the message loop, however, the only way to terminate the loop is to have the application's main window function post a WM_QUIT message in the application queue by using the **PostQuitMessage** function. When the **GetMessage** function retrieves a WM_QUIT message, it returns NULL, which terminates the message loop. By convention, only the window function for the application's main window ever posts a WM_QUIT message and then only when the the main window is being destroyed (that is, when the window function has received a WM_DESTROY message).

Although **WinMain** specifies a return-value type, Windows does not currently use the return value. While you are debugging an application, however, a return value can be helpful. In general, you may use the same return-code conventions that standard C programs use: zero for successful execution, nonzero for error. The **PostQuitMessage** function lets the window function specify the return value. This value is then copied to the **wParam** field of the WM_QUIT message. To return this value after terminating the message loop, use the following statement:

```
return (msg.wParam);   /* Returns the value from PostQuitMessage */
```

Although standard C programs typically clean up and free resources just prior to termination, Windows applications must be prepared to clean up as each window is destroyed. If you do not clean up as each window is destroyed, you may lose some data. For example, when Windows itself terminates, it destroys each window but does not return control to the application's message loop. This means that the loop never retrieves the WM_QUIT message and the statements after the loop are not executed. (Windows does send each application a message before terminating, so an application does have an opportunity to carry out tasks before terminating. See Chapter 11, "File Input and Output," for an illustration of the WM_QUERYENDSESSION message.)

3.4.10 The Initialization Function

Most applications use an initialization function to register their window classes and carry out work for initializing the instance. An initialization function is one means of keeping the **WinMain** function simple and readable, but it is also a means of organizing the initialization tasks so that they may be placed in a separate code segment and discarded after use. The Generic application does not discard its initialization function, but a sample application described later in this guide will.

The following example shows how to create an initialization function. To show how to use Windows memory-management functions, this example, rather than declaring a global structure, allocates temporary storage for the **WNDCLASS** structure.

```
BOOL GenericInit(hInstance)
HANDLE hInstance;                    /* current instance          */
{
    HANDLE hMemory;                  /* handle to allocated memory */
    PWNDCLASS pWndClass;             /* structure pointer          */
    BOOL bSuccess;                   /* RegisterClass() result     */

    hMemory = LocalAlloc(LPTR, sizeof(WNDCLASS));
    pWndClass = (PWNDCLASS) LocalLock(hMemory);

    pWndClass->style = NULL;
    pWndClass->lpfnWndProc = GenericWndProc;
    pWndClass->hInstance = hInstance;
    pWndClass->hIcon = LoadIcon(NULL, IDI_APPLICATION);
    pWndClass->hCursor = LoadCursor(NULL, IDC_ARROW);
    pWndClass->hbrBackground = GetStockObject(WHITE_BRUSH);
    pWndClass->lpszMenuName = (LPSTR) NULL;
    pWndClass->lpszClassName = (LPSTR) "Generic";

    bSuccess = RegisterClass(pWndClass);

    LocalUnlock(hMemory);                /* Unlocks the memory    */
    LocalFree(hMemory);                  /* Returns it to Windows */

    return (bSuccess); /* Returns result of registering the window */
}
```

3.4.11 Using Temporary Storage

Since the **RegisterClass** function copies your window-class information to an internal table, you do not need to preserve the window-class information after the class has been registered. This means you can allocate temporary storage for the class structure, register the class, then free the storage.

In Windows, you can allocate temporary storage by using the **LocalAlloc** function. The **LocalAlloc** function returns a handle (not a pointer) to the temporary storage. This is unlike the C run-time **malloc** function, which returns a pointer. Windows always identifies an allocated memory block with a handle. To use the memory, you must lock the memory block by using the **LocalLock** function. **LocalLock** returns a pointer to the memory block, and you may use this pointer just as you would the pointer returned by **malloc**.

After you have used a memory block, you must unlock it by using the **LocalUnlock** function. You should unlock it even if you plan to use it again. In general, lock a memory block just before using it and unlock it immediately after using it. You must unlock the memory block before freeing it with the **LocalFree** function.

3.4.12 The Application Command Line

You can examine the command line used to start your application by using the lpCmdLine parameter. The lpCmdLine parameter points to the start of a character array that contains the command exactly as it was typed by the user. Unlike C programs, the command line is not automatically separated into individual fields. If you wish to extract filenames or options from the command line, you need to provide the appropriate statements.

3.5 The Window Function

Every window must have a window function. The window function provides a response to input and window-management messages received from Windows. The window function can be a short function, processing only a message or two, or it may be complex, processing many types of messages for a variety of application windows.

A window function has the following form:

```
long FAR PASCAL GenericWndProc(hWnd, message, wParam, lParam)
HWND hWnd;                      /* window handle          */
unsigned message;               /* type of message        */
WORD wParam;                    /* additional information */
```

```
LONG lParam;                          /* additional information */
{
    switch (message) {
            .
            .
            .
        default:
    return (DefWindowProc(hWnd, message, wParam, lParam));
    }
    return (NULL);
}
```

The window function uses the **PASCAL** calling convention. Since Windows calls this function directly and always uses this convention, **PASCAL** is required. The window function also uses the **FAR** keyword in its definition since Windows uses a 32-bit address whenever it calls a function. Also, you must name the window function in an **EXPORTS** statement in the application's module-definition file.

The window function receives messages from Windows. These may be input messages that have been dispatched by the **WinMain** function or window-management messages that come directly from Windows. The window function must examine each message and either carry out some specific action based on the message or pass the message back to Windows for default processing through the **DefWindowProc** function.

The parameter, message, defines the message type. This parameter is used in a **switch** statement to direct processing to the correct case. The lParam and wParam parameters contain additional information about the message. The window function typically uses these parameters to carry out the requested action. If a window function doesn't process a message, it must pass it to the **DefWindowProc** function. This ensures that any special actions that affect the window, the application, or Windows itself can be carried out.

Most window functions process the WM_DESTROY message. Windows sends this message to the window function immediately after destroying the window. The message gives the window function the opportunity to finish its processing and, if it is the window function for the application's main window, to post a WM_QUIT message in the application queue. The following example shows how the main window function should process this message:

```
case WM_DESTROY:
    PostQuitMessage(NULL);
    break;
```

The **PostQuitMessage** function places a WM_QUIT message in the application's queue. When the **GetMessage** function retrieves this message, it will terminate the message loop and the application.

A window function receives messages from two sources: input messages from the message loop and window-management messages from Windows. Input messages correspond to mouse, keyboard, and, sometimes, timer input. Typical input messages are WM_KEYDOWN, WM_KEYUP, WM_MOUSEMOVE, and WM_TIMER, all of which correspond directly to hardware input.

Windows sends window-management messages directly to a window function without going through the application queue or message loop. These window messages are typically requests for the window function to carry out some action, such as painting its client area or supplying information about the window. The messages may also inform the window function of changes that Windows has made to the window. Some typical window-management messages are WM_CREATE, WM_DESTROY, and WM_PAINT.

The window function should return a long value. The actual value to be returned depends on the message received. For example, the correct return value for WM_DESTROY is NULL. If the window function doesn't process a message, it should return the **DefWindowProc** function's return value.

3.6 Creating an About Dialog Box

The *Microsoft Windows Application Style Guide* recommends that you include an About dialog box with every application. A dialog box is a temporary window that displays information or prompts for user input. The About dialog box displays the application name and copyright information. The user directs the application to display the About dialog box by choosing the About command from a menu. This menu is either the first menu in your application, or if there is no other menu in the application, the system menu.

You create and display a dialog box by using the **DialogBox** function. This function takes a dialog-box template, a procedure-instance address, and a handle to a parent window, and creates a dialog box through which you can display output and prompt for user input.

To display and use a dialog box, you need to carry out these steps:

1. Create a dialog-box template and add it to your resource script file.
2. Create a dialog function and add it to your C-language source file.
3. Export your dialog function in your module-definition file.
4. Append an About command to your system menu and process the WM_SYSCOMMAND message.

Once you have completed these steps, the user can choose the About command from the system menu to display the dialog box. The following sections explain the steps necessary to create an About dialog box.

3.6.1 Creating a Dialog-Box Template

A dialog-box template is a textual description of the dialog style, contents, shape, and size. You can create a template by hand or by using the Windows 2.0 Dialog Editor. In this example, the template is created by hand.

You create a dialog-box template by creating a resource script file. A resource script file contains definitions of resources to be used by the application, such as icons, cursors, and dialog-box templates. To create an About dialog-box template, you use a **DIALOG** statement and fill it with control statements, as shown in the following example:

```
AboutBox DIALOG 22, 17, 144, 75
STYLE WS_POPUP | WS_DLGFRAME
BEGIN
    CTEXT "Microsoft Windows"     -1, 0,  5, 144,  8
    CTEXT "Generic Application"   -1, 0, 14, 144,  8
    CTEXT "Version 1.0"           -1, 0, 34, 144,  8
    DEFPUSHBUTTON "OK"          IDOK, 53, 59, 32, 14, WS_GROUP
END
```

The **DIALOG** statement starts the dialog-box template. The name, AboutBox, identifies the template when the **DialogBox** function is used to create the dialog box. The box's upper-left corner is placed at the point (22,17) in the parent window's client area. The box is 144 units wide by 75 units high. Dialog-box width units are one quarter the width of the system-font characters. Dialog-box height units are one eighth the height of the system-font characters.

The **STYLE** statement defines the dialog-box style. This particular style is a pop-up window with a framed border, which is the typical style used for modal dialog boxes. The **BEGIN** and **END** statements mark the beginning and end of the control definitions. The dialog box contains text and a default push button. The push button lets the user send input to the dialog function to terminate the dialog box.

The statements, strings, and integers contained between the **BEGIN** and **END** statements describe the contents of the dialog box. You don't need to know the specifics of the numerical data since you would normally create this description by using Dialog Editor. **CTEXT** creates a rectangle with the quoted text centered in a rectangle. This statement appears several times for the various text that appears in the dialog box. **DEFPUSHBUTTON** creates a push button that allows the user to give a default response; in this case, to choose the "OK" button, causing the dialog box to disappear.

The statements in this file were created with a text editor, and were based on a dialog box used in another application. Many such resources can be copied from other applications and easily modified by using an editor. Dialog boxes can also be created from scratch by using Dialog Editor. The files created by Dialog Editor contain statements that are somewhat different from the statements shown here, and such files usually are edited only by using Dialog Editor. For more information about using Dialog Editor to create dialog boxes, see *Microsoft Windows Programming Tools.*

The WS_POPUP, WS_DLGFRAME, IDOK, and WS_GROUP constants used in the dialog-box template are defined in the Windows include file. You should include this file in the resource script file by using the **include** statement at the beginning of the file.

3.6.2 Creating an Include File

It is often useful to create an include file in which to define constants and function prototypes for your application. Most applications consist of at least two source files that share common constants: the C-language source file and the resource script file. Since the resource compiler, **rc**, carries out the same preprocessing as the C compiler, it is useful and convenient to place constant definitions in a single include file and then include that file in both the C-language source file and the resource script file.

For example, for the Generic application, you can place the function prototypes for the **WinMain**, GenericWndProc, About, and GenericInit functions, and the definition of the menu ID for the About command, in the include file, *generic.h.* The file should look like this:

```
#define ID_ABOUT 100

int PASCAL WinMain(HANDLE, HANDLE, LPSTR, int);
BOOL GenericInit(HANDLE);
long FAR PASCAL GenericWndProc(HWND, unsigned, WORD, LONG);
BOOL FAR PASCAL About(HWND, unsigned, WORD, LONG);
```

Since this include file includes Windows data types, you must include it after including the Windows include file. In other words, the beginning of your source files should look like this:

```
#include "windows.h"      /* required for all Windows applications */
#include "generic.h"      /* specific to this program              */
```

3.6.3 Creating a Dialog Function

The dialog function creates the About dialog box. The function that processes input for the dialog box is called About. The About function, like other dialog functions, uses the same parameters as a window function, but processes only messages that are specific to the dialog box. (About returns TRUE if it processes a message, and FALSE if it does not.) Unlike window functions, About usually processes only user-input messages, such as WM_COMMAND, and does not have to send unprocessed messages to the **DefWindowProc** function. The About function looks like this:

```
BOOL FAR PASCAL About(hDlg, message, wParam, lParam)
HWND hDlg;
unsigned message;
WORD wParam;
LONG lParam;
{
    switch (message) {
        case WM_INITDIALOG:      /* message: initialize dialog box */
            return (TRUE);

        case WM_COMMAND:            /* message: received a command */
            if (wParam == IDOK) {          /* "OK" box selected?   */
                EndDialog(hDlg, NULL);     /* Exits the dialog box */
                return (TRUE);
            }
            break;
    }
    return (FALSE);                     /* Didn't process a message */
}
```

The dialog function, like the window function, uses the **PASCAL** calling convention. Since Windows calls this function directly and always uses this convention, **PASCAL** is required. The dialog function also uses the **FAR** keyword in its definition since Windows uses a 32-bit address whenever it calls a function. Also, you must name the dialog function in an **EXPORTS** statement in the application's module-definition file. As with a window function, you must not call a dialog function directly from your application.

The About function processes two messages: WM_COMMAND and WM_INITDIALOG. The WM_INITDIALOG message is sent to a dialog function by Windows to let the function prepare before displaying the dialog box. In this case, WM_INITDIALOG returns TRUE so that the "focus" will be passed to the first control in the dialog box that has the WS_TABSTOP bit set (this will be the default push button). If WM_INITDIALOG had returned FALSE, the focus would not have been set to any control in the dialog box. In contrast to WM_INITDIALOG messages, WM_COMMAND messages are a result of user input. About responds to input to the "OK" button by calling the **EndDialog** function,

which directs Windows to remove the dialog box and continue execution of the application. The **EndDialog** function is used to terminate dialog boxes.

3.6.4 Appending About to the System Menu

Now that you have an About dialog box, you need some way to let the user tell you when to display it. In a simple application, like Generic, you can append an About command to the end of the system menu. To do this you must use the **GetSystemMenu** and **ChangeMenu** functions.

A convenient place to make this change is in the window function in response to the WM_CREATE message. Just as you can carry out work immediately after your window has been destroyed, you can also carry out work just as your window is created. Windows sends the WM_CREATE message to your window function immediately after creating the window and before the **CreateWindow** function returns control to your **WinMain** function.

The following example shows how to change the system menu for a window:

```
case WM_CREATE:          /* message: window being created */

   hMenu = GetSystemMenu(hWnd, FALSE);

   ChangeMenu(hMenu,                  /* menu handle          */
       NULL,                          /* menu item to change */
       NULL,                          /* new menu item        */
       NULL,                          /* menu identifier      */
       MF_APPEND | MF_SEPARATOR);     /* type of change       */

   ChangeMenu(hMenu,                  /* menu handle          */
       NULL,                          /* menu item to change */
       "A&bout Generic...",           /* new menu item        */
       ID_ABOUT,                      /* menu identifier      */
       MF_APPEND | MF_STRING);        /* type of change       */
   break;
```

The **GetSystemMenu** function retrieves the handle of the system menu for the given window. Each window has a private copy of the system menu that it can modify if necessary. The FALSE parameter ensures that the function will retrieve the current system menu, not a fresh copy.

The first **ChangeMenu** function appends a separator (a horizontal bar) to the end of the system menu. The second **ChangeMenu** function appends the command name "About Generic..." immediately after the separator. The first argument in each call specifies the menu to be changed. The last argument specifies the type of change. The other arguments specify additional information, such as the name of the command or the menu identifier (menu ID).

Notice the ampersand (&) in the "A&bout Generic..." string. This character immediately precedes the command mnemonic. A mnemonic is a unique letter or digit with which the user can access a menu or command. It is part of Windows' direct-access method—if a user presses the key for the mnemonic, Windows automatically selects the menu or chooses the command. In the case of "A&bout Generic...", Windows removes the ampersand and places an underscore under the letter "b" when displaying the menu.

Once the changes are made, the user will see the separator and the About command the next time he or she selects the system menu. If the user chooses the About command, Windows sends the window function a WM_SYSCOMMAND message containing the About command's menu ID; in this case, ID_ABOUT.

3.6.5 Processing the WM_SYSCOMMAND Message

Now that you've added a command to the system menu, you need to process the WM_SYSCOMMAND message. Windows sends this message to the window function when the user chooses a command from the system menu. Windows passes the menu ID identifying the command in the wParam parameter, so you can check to see which command was chosen. You can check the parameter by using a **switch** statement. In this case, you want to display the dialog box if the parameter is equal to ID_ABOUT, the About command's menu ID. For any other value, you must pass the message on to the **DefWindowProc** function. If you do not, you effectively disable all other commands the system menu.

The WM_SYSCOMMAND case should look like this:

```
FARPROC lpProcAbout;
    .
    .
    .
     case WM_SYSCOMMAND: /* message: command from system menu */
         if (wParam == ID_ABOUT) {
            lpProcAbout = MakeProcInstance(About, hInst);

            DialogBox(hInst,        /* current instance          */
                "AboutBox",         /* resource to use           */
                hWnd,               /* parent handle             */
                lpProcAbout);       /* About() instance address  */

             FreeProcInstance(lpProcAbout);
             break;
         }

         else                   /* Lets Windows process it       */
             return (DefWindowProc(hWnd, message,
                 wParam, lParam));
```

To display the dialog box, you need the procedure-instance address of the dialog function. You create the procedure-instance address by using the **MakeProcInstance** function. This function binds the data segment of the current application instance to a function pointer. This guarantees that when Windows calls the dialog function, the dialog function will use the data in the current instance and not some other instance of the application. **MakeProcInstance** returns the address of the procedure instance. This value should be assigned to a pointer variable that has the **FARPROC** type.

The **DialogBox** function creates the dialog box. It requires the current application's instance handle and the name of the dialog-box template. It uses this information to load the dialog-box template from the executable file. **DialogBox** also requires the handle of the parent window (the window to which the dialog box belongs) and the procedure-instance address.

The **DialogBox** function creates and displays the dialog box. The function does not return control until the user has closed the dialog box. Typically, the dialog box contains at least a push-button control to permit the user to close the box.

When the **DialogBox** function returns, the procedure-instance address of the dialog function is no longer needed, so the **FreeProcInstance** function frees the address. This invalidates the content of the pointer variable, making it an error to attempt to use the value again.

3.7 Creating a Module-Definition File

Every Windows application needs a module-definition file. This file defines the name, segments, memory requirements, and exported functions of the application. For a simple application, like Generic, you need at least the **NAME**, **STACKSIZE**, **HEAPSIZE**, and **EXPORTS** statements. However, most applications include a complete definition of the module, as shown in the following example:

```
;module-definition file for Generic -- used by link4.exe

NAME     Generic        ; application's module name

DESCRIPTION 'Sample Microsoft Windows Application'

STUB     'WINSTUB.EXE' ; Generates error message if application
                       ; is run without Windows

CODE     MOVEABLE      ; code can be moved in memory

;DATA must be MULTIPLE if program can be invoked more than once

DATA     MOVEABLE MULTIPLE
```

```
HEAPSIZE  1024
STACKSIZE 4096  ; recommended minimum for Windows applications

; All functions that will be called by any Windows routine
; MUST be exported.

EXPORTS
    GenericWndProc   @1  ; name of window-processing function
    About            @2  ; name of About processing function
```

Note

The semicolon is the delimiter for comments in the module-definition file; thus, all text following the semicolon is a programming comment.

The **NAME** statement defines the name of the application. This name is used by Windows to identify the application. The **NAME** statement is required.

The **DESCRIPTION** statement is an optional statement that places the message "Sample Microsoft Windows Application" in the application's executable file. This statement is typically used to add version control or copyright information to the file.

The **STUB** statement specifies another optional file that defines the Windows executable stub to be placed at the beginning of the file. This executable stub displays a warning message and terminates the application if the user attempts to run it without Windows.

The **CODE** statement defines the memory requirements of the application's code segment. The code segment contains the executable code that is generated when the *generic.c* file is compiled. Generic is a small-model application with only one code segment, which is defined as **MOVEABLE**. If the application is not running and Windows needs additional space in memory, Windows can move the code segment to make room for other segments.

The **DATA** statement defines the memory requirements of the application's data segment. The data segment contains storage space for all the static variables declared in the *generic.c* file. It also contains space for the program stack and local heap. The data segment, like the code segment, is **MOVEABLE**. The **MULTIPLE** keyword directs Windows to create a new data segment for the application each time the user starts a new instance of the application. Although all instances share the same code segment, each has its own data segment. Applications must have the **MULTIPLE** keyword.

The **HEAPSIZE** statement defines the size, in bytes, of the application's local heap. Generic uses its heap to allocate the temporary structure used to register the window class, so it specifies 1024 bytes of storage. Applications that use the local heap frequently should specify larger amounts of memory.

The **STACKSIZE** statement defines the size, in bytes, of the application's stack. The stack is used for temporary storage of function arguments. Any application, like Generic, that calls its own local function must have a stack. Generic specifies 4096 bytes of stack storage, the recommended minimum for a Windows application.

The **EXPORTS** statement defines the names and ordinal values of the functions to be exported by the application. Generic exports its window function, GenericWndProc, which has ordinal value 1 (this is an identifier; it could be any integer, but usually such values are assigned sequentially as the exports are listed). Windows requires that all functions, except **WinMain**, that are to be called by Windows must be exported in this way. These functions are referred to as "callback" functions. All window functions, whether for parent or child application windows, or for dialog boxes, are callback functions.

3.8 Putting Generic Together

At this point you are ready to put the sample application, Generic, together. You need to do the following:

1. Create the C-language source file.
2. Create the resource script file..
3. Create the module-definition file.
4. Create the **make** file.
5. Run the **make** file to compile and link the application.

The following sections describe each step.

3.8.1 Create the C-Language Source File

The C-language source file must contain the **WinMain** function, the GenericWndProc window function, the About dialog function, and the GenericInit initialization function. Name the file *generic.c* and make sure it looks like this:

```
#include "windows.h"      /* required for all Windows applications */
#include "generic.h"      /* specific to this program              */

HANDLE hInst;                                   /* current instance */

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;                  /* current instance             */
HANDLE hPrevInstance;              /* previous instance            */
LPSTR lpCmdLine;                   /* command line                 */
int nCmdShow;                      /* show-window type (open/icon) */
{
    HWND hWnd;                                      /* window handle */
    MSG msg;                                        /* message       */


    if (!hPrevInstance)       /* Has application been initialized? */
        if (!GenericInit(hInstance))
            return (NULL);    /* Exits if unable to initialize     */

    hInst = hInstance;        /* Saves the current instance        */

    hWnd = CreateWindow("Generic",                  /* window class     */
        "Generic Sample Application",               /* window name      */
        WS_OVERLAPPEDWINDOW,                        /* window style     */
        CW_USEDEFAULT,                              /* x position       */
        CW_USEDEFAULT,                              /* y position       */
        CW_USEDEFAULT,                              /* width            */
        CW_USEDEFAULT,                              /* height           */
        NULL,                                       /* parent handle    */
        NULL,                                       /* menu or child ID */
        hInstance,                                  /* instance         */
        NULL);                                      /* additional info  */

    if (!hWnd)                            /* Was the window created? */
        return (NULL);

    ShowWindow(hWnd, nCmdShow);           /* Shows the window        */
    UpdateWindow(hWnd);                   /* Sends WM_PAINT message  */

    while (GetMessage(&msg,         /* message structure             */
            NULL,         /* handle of window receiving the message */
            NULL,                 /* lowest message to examine     */
            NULL))                /* highest message to examine    */
        {
        TranslateMessage(&msg);     /* Translates virtual key codes */
        DispatchMessage(&msg);      /* Dispatches message to window */
    }
    return (msg.wParam); /* Returns the value from PostQuitMessage */
}

BOOL GenericInit(hInstance)
HANDLE hInstance;                    /* current instance           */
```

```
{
    HANDLE hMemory;                 /* handle to allocated memory */
    PWNDCLASS pWndClass;            /* structure pointer          */
    BOOL bSuccess;                  /* RegisterClass() result     */

    hMemory = LocalAlloc(LPTR, sizeof(WNDCLASS));
    pWndClass = (PWNDCLASS) LocalLock(hMemory);

    pWndClass->style = NULL;
    pWndClass->lpfnWndProc = GenericWndProc;
    pWndClass->hInstance = hInstance;
    pWndClass->hIcon = LoadIcon(NULL, IDI_APPLICATION);
    pWndClass->hCursor = LoadCursor(NULL, IDC_ARROW);
    pWndClass->hbrBackground = GetStockObject(WHITE_BRUSH);
    pWndClass->lpszMenuName = (LPSTR) NULL;
    pWndClass->lpszClassName = (LPSTR) "Generic";

    bSuccess = RegisterClass(pWndClass);

    LocalUnlock(hMemory);               /* Unlocks the memory    */
    LocalFree(hMemory);                 /* Returns it to Windows */

    return (bSuccess); /* Returns result of registering the window */
}

BOOL GenericInit(hInstance)
HANDLE hInstance;                   /* current instance           */
{
    HANDLE hMemory;                 /* handle to allocated memory */
    PWNDCLASS pWndClass;            /* structure pointer          */
    BOOL bSuccess;                  /* RegisterClass() result     */

    hMemory = LocalAlloc(LPTR, sizeof(WNDCLASS));
    pWndClass = (PWNDCLASS) LocalLock(hMemory);

    pWndClass->style = NULL;
    pWndClass->lpfnWndProc = GenericWndProc;
    pWndClass->hInstance = hInstance;
    pWndClass->hIcon = LoadIcon(NULL, IDI_APPLICATION);
    pWndClass->hCursor = LoadCursor(NULL, IDC_ARROW);
    pWndClass->hbrBackground = GetStockObject(WHITE_BRUSH);
    pWndClass->lpszMenuName = (LPSTR) NULL;
    pWndClass->lpszClassName = (LPSTR) "Generic";

    bSuccess = RegisterClass(pWndClass);

long FAR PASCAL GenericWndProc(hWnd, message, wParam, lParam)
HWND hWnd;                          /* window handle              */
unsigned message;                   /* type of message            */
WORD wParam;                        /* additional information     */
LONG lParam;                        /* additional information     */
{
    FARPROC lpProcAbout;           /* pointer to the About function */
    HMENU hMenu;                   /* handle to the System menu     */

    switch (message) {
        case WM_SYSCOMMAND: /* message: command from system menu */
            if (wParam == ID_ABOUT) {
                lpProcAbout = MakeProcInstance(About, hInst);
```

```
            DialogBox(hInst,          /* current instance         */
                "AboutBox",           /* resource to use          */
                hWnd,                 /* parent handle            */
                lpProcAbout);         /* About() instance address */

            FreeProcInstance(lpProcAbout);
            break;
        }

        else                    /* Lets Windows process it       */
            return (DefWindowProc(hWnd, message,
                wParam, lParam));

    case WM_CREATE:             /* message: window being created */

        hMenu = GetSystemMenu(hWnd, FALSE);

        ChangeMenu(hMenu,                   /* menu handle         */
            NULL,                           /* menu item to change */
            NULL,                           /* new menu item       */
            NULL,                           /* menu identifier     */
            MF_APPEND | MF_SEPARATOR);      /* type of change      */

        ChangeMenu(hMenu,                   /* menu handle         */
            NULL,                           /* menu item to change */
            "A&bout Generic...",            /* new menu item       */
            ID_ABOUT,                       /* menu identifier     */
            MF_APPEND | MF_STRING);         /* type of change      */
        break;

    case WM_DESTROY:          /* message: window being destroyed */
        PostQuitMessage(0);
        break;

    default:                  /* Passes it on if unprocessed     */
        return (DefWindowProc(hWnd, message, wParam, lParam));
    }
    return (NULL);
}
```

3.8.2 Create the Resource Script File

The resource script file must contain the dialog-box template for the About dialog box. Name the file *generic.rc* and make sure it looks like this:

```
#include "windows.h"
#include "generic.h"

AboutBox DIALOG 22, 17, 144, 75
STYLE WS_POPUP | WS_DLGFRAME
BEGIN
    CTEXT "Microsoft Windows"     -1, 0,  5, 144,  8
    CTEXT "Generic Application"   -1, 0, 14, 144,  8
    CTEXT "Version 1.0"           -1, 0, 34, 144,  8
    DEFPUSHBUTTON "OK"          IDOK, 53, 59, 32, 14, WS_GROUP
END
```

3.8.3 Create the Module-Definition File

The module-definition file must contain the module definitions for Generic. Name the file *generic.def* and make sure it looks like this:

```
;module-definition file for Generic -- used by link4.exe

NAME    Generic       ; application's module name

DESCRIPTION 'Sample Microsoft Windows Application'

STUB    'WINSTUB.EXE' ; Generates error message if application
                      ; is run without Windows

CODE    MOVEABLE      ; code can be moved in memory

;DATA must be MULTIPLE if program can be invoked more than once

DATA    MOVEABLE MULTIPLE

HEAPSIZE  1024
STACKSIZE 4096  ; recommended minimum for Windows applications

; All functions that will be called by any Windows routine
; MUST be exported.

EXPORTS
    GenericWndProc   @1  ; name of window-processing function
    About            @2  ; name of About processing function
```

3.8.4 Create a Make File

Once you have the source files, you can create Generic's **make** file, then compile and link the application by using the **make** program. To compile and link Generic, the **make** file must follow these steps:

- Use the Microsoft C compiler, **cl**, to compile the *generic.c* file.
- Use the Windows linker, **link4**, to link the *generic.obj* object file with the Windows library and the module-definition file, *generic.def*.
- Use the resource compiler, **rc**, to create a binary resource file and add it to the executable file of the Windows application.

The following will properly compile and link the files created for Generic:

```
# Update the resource if necessary

generic.res: generic.rc generic.h
    rc -r generic.rc

# Update the object file if necessary
```

```
generic.obj: generic.c generic.h
    cl -c -Gsw -Zp generic.c

# Update the executable file if necessary, and if so, add the resource back in.
# The /NOE must be included when linking with Windows libraries.

generic.exe: generic.obj generic.def
    link4 generic, , , slibw/NOE, generic.def
    rc generic.res

# If the .res file is new and the .exe file is not, update the resource.
# Note that the .rc file can be updated without having to either
# compile or link the file.

generic.exe: generic.res
    rc generic.res
```

The first two lines direct **make** to create a compiled resource file, *generic.res*, if the resource script file, *generic.rc*, or the new include file, *generic.h*, has been updated. The **–r** option of the **rc** command creates a compiled resource file without attempting to add it to an executable file.

The next two lines direct **make** to create the *generic.obj* file if *generic.c* or *generic.h* has a more recent access date than the current *generic.obj* file. The **cl** command takes several command-line options that prepare the application for execution under Windows. The minimum required options are **–c**, **–Gw**, **–Zp**. In this case, **cl** "assumes" that Generic is a small-model application. Generic and all other applications in the *Microsoft Windows Programmer's Learning Guide* are small-model applications.

The **make** program then creates the *generic.exe* file if any one of the *generic.obj*, *generic.def*, or *generic.res* files has a more recent access date than the current *generic.exe* file. Small Windows applications, like Generic, must be linked with the Windows *slibw.lib* library. The **link4** program will also link with the C run-time libraries, *slibc.lib* and *libh.lib*, by default. The object file, *generic.obj*, and the module-definition file, *generic.def*, are used as arguments in the **link4** command line.

The last **rc** command automatically appends the compiled resources in the *generic.res* file to the executable file, *generic.exe*.

3.9 Using Generic as a Template

Generic, though it has no functions, provides certain essentials that make it an appropriate starting point for your applications. It conforms to the standards given in the *Microsoft Windows Application Style Guide* for appearance and cooperation with other applications. It contains all the files an application can have: *.def*, *.h*, *.rc*, *.c*, and **make**. The About dialog box, an application standard, is included, as well as the About Generic... command on the System menu. This is a *Microsoft Windows Application*

Style Guide standard for applications without application menus. (Applications with application menus should place the About Generic... command as the last item on the first menu.)

You can use Generic as a template to build your own applications. To do this, you copy and rename the sources of an existing application, such as Generic, then change relevant function names, and insert new code. All sample applications in this guide have been created by copying and renaming Generic's source files, then modifying some of the function and resource names to make them unique to each new application.

The following procedure describes how to use Generic as a template, allowing you to adapt its source files to your application:

1. Choose your application's filename.
2. Copy the following Generic source files, renaming them to match your application's filename: *generic.c*, *generic.h*, *generic.def*, *generic.rc*, and *generic*.
3. Use a text editor to change each occurrence of "Generic" in your application's C-language source file (formerly *generic.c*) to your application's name. This includes changing the following:
    - The window-function name: GenericWndProc
    - The initialization-function name: GenericInit
    - The class name: Generic
    - The window title: Generic Sample Application
    - The include filename: *generic.h*
4. Use a text editor to change each occurrence of "Generic" in your application's module-definition file (formerly *generic.def*) to your application's name. This includes changing the following:
    - The application name: Generic
    - The window-function name: GenericWndProc
5. Use a text editor to change each occurrence of "Generic" in your application's resource script file (formerly *generic.rc*) to your application's name. This includes changing the following:
    - The include filename: *generic.h*
    - The application title: Generic Application

6. Use a text editor to change each occurrence of "Generic" in your application's **make** file (formerly *generic*) to your application's name. This includes changing the following:
    - The C-language source filename: *generic.c*
    - The object filename: *generic.obj*
    - The executable filename: *generic.exe*
    - The module-definition filename: *generic.def*

As you add new functions, resources, and include files to your applications, be sure to use your application's filename to ensure that these names are unique.

Chapter 4
Output to a Window

4.1 Introduction

In Microsoft Windows, all output to a window is performed by the graphics device interface (GDI). This chapter explains how to use the GDI functions to draw within the client area of a window. In particular, it describes how to draw lines and figures, write text, and create pens and brushes. It also describes the painting and drawing process and explains the purpose of the display context and the WM_PAINT message.

4.2 The Display Context

To draw within a window, all you need is the handle to the window. You use this handle to retrieve a handle to the display context of the window's client area. A display context defines the output device and the current drawing tools, colors, and other drawing information used by GDI to generate output. All GDI output functions require a display-context handle. No output can be performed without one.

For output to a window, Windows requires that you retrieve a handle to the display context for that window. The method you use to retrieve a handle depends on where you plan to perform the output operations. Although you can draw and write anywhere in an application, including within the **WinMain** function, most applications do so only in the window function. Although the window function can draw within the client area in response to almost any message, the most common time to draw and write is in response to a WM_PAINT message. Windows sends this message to a window function when changes to the window may have altered the content of the client area. Since only the application knows what is in the client area, Windows sends the message to the window function so it can restore the client area.

If you plan to draw within the client area at any time other than in response to a WM_PAINT message, you must use the **GetDC** function to retrieve the handle to the display context. For the WM_PAINT message, you must use the **BeginPaint** function.

Whenever you retrieve a display context for a window, the context is only on temporary loan from Windows to your application. A display context is a shared resource and as long as one application has it, no other application can retrieve it. This means you must release the display context as soon as possible after using it to draw within the window. If you retrieve a display context by using the **GetDC** function, you must use the **ReleaseDC** function to release it. Similarly, for **BeginPaint**, you use the **EndPaint** function.

4.2.1 Using the GetDC Function

You typically use the **GetDC** function to provide instant feedback to some action by the user, such as drawing a line as the user moves the mouse cursor (pointer) through the window. The function returns a display-context handle that you can use in any GDI output function. The following example shows how to use the **GetDC** function to retrieve a display-context handle and write the string "Hello Windows!" in the client area:

```
hDC = GetDC(hWnd);
TextOut(hDC, 10,10, "Hello Windows!", 14);
ReleaseDC(hWnd, hDC);
```

In this example, the **GetDC** function returns the display context for the window identified by the hWnd parameter, and the **TextOut** function writes the string at the point (10,10) in the window's client area. The **ReleaseDC** function releases the display context.

Anything you draw in the client area will be erased the next time Windows sends a WM_PAINT message to the window function. The reason for this is that Windows sends a WM_ERASEBKGND message to the window function as part of the WM_PAINT message processing. If you pass WM_ERASEBKGND on to the **DefWindowProc** function, **DefWindowProc** fills the client area by using the class background brush, completely erasing any output you may have previously drawn there.

4.2.2 The WM_PAINT Message

Windows posts a WM_PAINT message when some operation by the user has changed the window. For example, Windows posts a WM_PAINT message when the user closes a window that covers part of another window. Since a window shares the screen with other windows, anything the user does in one window can have an impact on the content and appearance of another window. However, you can do nothing about the change until your application receives the WM_PAINT message.

Windows posts a WM_PAINT message by making it the last message in the application queue. This means any input is processed before the WM_PAINT message. In fact, the **GetMessage** function also retrieves any input generated after the WM_PAINT message is posted. That is, **GetMessage** retrieves the WM_PAINT message from the queue only when there are no other messages. The reason for this is to let the application carry out any operations that might affect the appearance of the window. In general, output operations should be carried out as infrequently as possible to avoid flicker and other distracting effects. Windows guarantees this by holding the WM_PAINT message until it is the last message.

The following example shows how to process a WM_PAINT message:

```
PAINTSTRUCT ps;
    .
    .
    .
case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    /* Output operations */
    EndPaint(hWnd, &ps);
    break;
```

The **BeginPaint** and **EndPaint** functions are required. **BeginPaint** fills the **PAINTSTRUCT** structure, ps, with information about the paint request, such as the part of the client area that needs redrawing, and returns a handle to the display context. You can use the handle in any GDI output functions. The **EndPaint** function ends the paint request and releases the display context.

You must not use the **GetDC** and **ReleaseDC** functions in place of the **BeginPaint** and **EndPaint** functions. **BeginPaint** and **EndPaint** carry out special tasks, such as validating the client area and sending the WM_ERASEBKGND message, that ensure that the paint request is processed properly. If you use **GetDC** instead of **BeginPaint**, the painting request will never be satisfied and your window function will continue to receive the same paint request.

4.2.3 Invalidating the Client Area

Windows is not the only source of WM_PAINT messages. You can also generate WM_PAINT messages for your windows by using two functions: **InvalidateRect** and **InvalidateRgn**. These functions mark all or part of a client area as invalid (in need of redrawing). For example, the following function invalidates the entire client area:

```
InvalidateRect(hWnd, NULL, TRUE);
```

This example invalidates the entire client area for the window identified by the hWnd parameter. The NULL argument, used in place of a rectangle structure, specifies the entire client area. The TRUE argument causes the background to be erased.

When the client area is marked as invalid, Windows posts a WM_PAINT message. If other parts of the client area are marked as invalid, Windows does not post another WM_PAINT message. Instead, it adds the invalidated areas to the previous area, so that all areas are processed by the same WM_PAINT message.

If you change your mind about redrawing the client area, you can validate parts of it by using the **ValidateRect** and **ValidateRgn** functions. These functions remove any previous invalidation and may remove the WM_PAINT message if no other invalidated area remains.

If you do not want to wait for the WM_PAINT message to be retrieved from the application queue, you can force an immediate WM_PAINT message by using the **UpdateWindow** function. If there is any invalid part of the client area, **UpdateWindow** pulls the WM_PAINT message for the given window from the queue and sends it directly to the window function.

4.2.4 Display Context and Device Context

A display context is actually a type of device context that has been especially prepared for output to the client area of a window. A device context defines the device, drawing tools, and drawing information for a complete device, such as a display or printer; a display context defines these things only for a window's client area. To prepare a display context, Windows adjusts the device origin so that it aligns with the upper-left corner of the client area instead of with the upper-left corner of the display. It also sets a clipping rectangle so that output to a display context is "clipped" to the client area. This means any output that would otherwise appear outside the client area is not sent to the display. Although you can retrieve a device context for the entire display, you should avoid doing so since it overrides Windows' careful control of the shared display.

4.2.5 The Coordinate System

The default coordinate system for a display context is very simple. The upper-left corner of the client area is the origin, or point (0,0). Each pixel to the right represents one unit along the positive *x*-axis. Each pixel down represents one unit along the positive *y*-axis.

You can modify this coordinate system by changing the mapping mode and display origins. The mapping mode defines the coordinate-system units. The default mode is MM_TEXT, or one pixel per unit. You can also specify mapping modes that use inches or millimeters as units. The origin of the coordinate system can be moved to any point.

For simplicity, the examples in this chapter and throughout this guide use the default coordinate system.

4.3 Creating, Selecting, and Deleting Drawing Tools

GDI lets you use a variety of drawing tools to draw within a window. GDI provides pens to draw lines, brushes to fill interiors, and fonts to write text. To use these tools, you create them by using functions such as **CreatePen** and **CreateSolidBrush**, then select them into the display context by using the **SelectObject** function. When you are done using a drawing tool, you can delete it by using the **DeleteObject** function.

You can create a pen for drawing lines and borders by using the **CreatePen** function. The function returns a handle to a pen that has the specified style, width, and color. The following example creates a dashed, black pen, one pixel wide:

```
HPEN hDashPen;
    .
    .
    .

hDashPen = CreatePen(1, 1, RGB(0, 0, 0));
```

The **RGB** utility creates a 32-bit value representing a red, green, and blue color value. The three arguments specify the intensity of the colors red, green, and blue, respectively. In this example, all colors have zero intensity, so the specified color is black.

You can create solid brushes for drawing and filling by using the **CreateSolidBrush** function. This function returns a handle to a brush that contains the specified solid color. The following example shows how to create a red brush:

```
HBRUSH hRedBrush
    .
    .
    .

hRedBrush = CreateSolidBrush(RGB(255, 0, 0));
```

Once you have created a drawing tool, you can select it into a display context by using the **SelectObject** function. The following example selects the red brush for drawing:

```
HBRUSH hOldBrush;
    .
    .
    .
hOldBrush = SelectObject(hDC, hRedBrush);
```

In this example, **SelectObject** returns a handle to the previous brush. In general, you should save the handle of the previous drawing tool so that you can restore it later.

You do not have to create or select a drawing tool before using a display context. Windows provides default drawing tools with each display context; for example, a black pen, a white brush, and the system font.

You can delete drawing objects you no longer need by using the **DeleteObject** function. The following example deletes the brush identified by the handle, hRedBrush:

```
DeleteObject(hRedBrush);
```

You must not delete a selected drawing tool. If necessary, you can use the **SelectObject** function to restore a previous drawing tool and remove the tool to be deleted from the selection, as shown in the following example:

```
SelectObject(hDC, hOldBrush);
DeleteObject(hRedBrush);
```

Although you can create and select fonts for writing text, working with fonts is a fairly involved process and is not described in this chapter. For a full discussion of how to create and select fonts, see Appendix A, "Fonts."

4.4 Drawing and Writing

GDI provides a wide variety of output operations, from drawing lines to writing text. Specifically, you can use the **LineTo**, **Rectangle**, **Ellipse**, **Arc**, **Pie**, **TextOut**, and **DrawText** functions to draw lines, rectangles, circles, arcs, pie wedges, and text. In all cases, the functions use the selected pen and brush to draw borders and fill interiors, and the selected font to write text.

You can draw lines by using the **LineTo** function. You usually combine the **MoveTo** and **LineTo** functions to draw lines. The following example draws a line from the point (10,90) to the point (360,90):

```
MoveTo(hDC, 10, 90);
LineTo(hDC, 360, 90);
```

You can draw a rectangle by using the **Rectangle** function. This function uses the selected pen to draw the border, and the selected brush to fill the interior. The following example draws a rectangle that has its upper-left and lower-right corners at the points (10,30) and (60,80), respectively:

```
Rectangle (hDC, 10, 30, 60, 80);
```

You can draw an ellipse or circle by using the **Ellipse** function. The function uses the selected pen to draw the border, and the selected brush to fill the interior. The following example draws an ellipse that is bounded by the rectangle specified by the points (160,30) and (210,80):

```
Ellipse (hDC, 160, 30, 210, 80);
```

You can draw arcs by using the **Arc** function. You draw an arc by defining the circle, then specifying on which points the arc starts and ends. The following example shows an arc drawn from the point (10,90) to the point (360,90):

```
Arc(hDC, 10, 90, 360, 120, 10, 90, 360, 90);
```

You can draw a pie wedge by using the **Pie** function. A pie wedge consists of an arc and two radii extending from the focus of the arc to its respective endpoints. The **Pie** function uses the selected pen to draw the border, and the selected brush to fill the interior. The following example draws a pie wedge that is bounded by the rectangle specified by the points (310,30) and (360,80) and that starts and ends at the points (360,30) and (360,80), respectively:

```
Pie (hDC, 310, 30, 360, 80, 360, 30, 360, 80);
```

You can display text by using the **TextOut** function. The function displays a string starting at the specified point. The following example displays the string "A Sample String" at the point (1,1):

```
TextOut(hDC, 1, 1, "A Sample String", 15);
```

You can also display text by using the **DrawText** function. This function is similar to **TextOut**, except that it lets you write text on multiple lines. The following example displays the string "This long string illustrates the DrawText function" on multiple lines in the specified rectangle:

```
RECT rcTextBox;
PSTR pText = "This long string illustrates the DrawText function";
    .
    .
    .

SetRect(TextBox, 1, 10, 160, 40);
DrawText(hDC, pText, strlen(pText), rcTextBox, DT_LEFT);
```

This example displays the string pointed to by the pText parameter as one or more left-aligned lines in the rectangle specified by the points (1,10) and (160,40).

Although you can also create and display bitmaps in a window, the process is not described in this chapter. For full details, see Chapter 9, "Bitmaps."

4.5 Computing a String's Length

You may use the C run-time **strlen** function in your applications to determine the length of a string, but you must be careful to provide the correct arguments. In small-model applications, such as the following Output, any C run-time string functions you may use expect near pointers (16-bit addresses). In some cases, however, string variables may have the **LPSTR** type, meaning their addresses are passed as 32-bit values. You could potentially cast the long address to a short one by using the **PSTR** type, but you would need to ensure that the string was in the application's data segment. The easiest solution is to replace the C run-time function with the locally defined _lstrlen function that processes 32-bit addresses. The following example shows just such a function:

```
int _lstrlen(lpString)
LPSTR lpString;
{
    int i;
    for (i = 0; *lpString++; ++i);
    return(i);
}
```

This function takes a pointer to a null-terminated string and returns an integer count of the number characters.

4.6 A Sample Application: Output

This sample application illustrates how to use the WM_PAINT message to draw within the client area, as well as how to create and use drawing tools. The Output application is a simple extension of the Generic application described in the previous chapter. To create the Output application, copy and rename the source files of the Generic application, then make the following modifications:

1. Add new variables.
2. Modify the WM_CREATE case.
3. Add a WM_PAINT case.

4. Modify the WM_DESTROY case.
5. Add an _lstrlen function.
6. Compile and link the application.

This sample assumes that you have a color display. If you do not, GDI will simulate some of the color output by "dithering." Dithering is a method of simulating a color by creating a unique pattern with two available colors; for example, simulating green by using black and white pixels.

4.6.1 Add New Variables

You need several new global variables for this sample application. Add the following variables at the beginning of your C-language source file:

```
HANDLE hInst;

HPEN hDashPen;                          /* "---" pen handle    */
HPEN hDotPen;                           /* "..." pen handle    */
HBRUSH hOldBrush;                       /* old brush handle    */
HBRUSH hRedBrush;                       /* red brush handle    */
HBRUSH hGreenBrush;                     /* green brush handle */
HBRUSH hBlueBrush;                      /* blue brush handle   */
```

You also need new local variables in the window function. Declare the following at the beginning of the function:

```
HDC hDC;                          /* display-context variable  */
PAINTSTRUCT ps;                   /* paint structure           */
LPSTR lpText = "Hello Windows!  This is a very long line indeed.";
RECT rcTextBox;                   /* rectangle around the text */
HPEN hOldPen;                     /* old pen handle            */
```

4.6.2 Modify the WM_CREATE Case

You need to create the drawing tools to be used in Output's client area before any drawing is carried out. Since you need to create these tools only once, a convenient place to do so is in the WM_CREATE message. Modify the WM_CREATE case so it looks like this:

```
case WM_CREATE:

    hRedBrush =   CreateSolidBrush(RGB(255,   0,   0));
    hGreenBrush = CreateSolidBrush(RGB(  0, 255,   0));
    hBlueBrush =  CreateSolidBrush(RGB(  0,   0, 255));

    /* Create the "---" pen */

    hDashPen = CreatePen(1,                          /* style */
        1,                                           /* width */
        RGB(0, 0, 0));                               /* color */
```

```
    /* Create the "..." pen */

    hDotPen = CreatePen(2,                          /* style */
        1,                                          /* width */
        RGB(0, 0, 0));                              /* color */

    hMenu = GetSystemMenu(hWnd, FALSE);
    ChangeMenu(hMenu, NULL, NULL, NULL, MF_APPEND | MF_SEPARATOR);
    ChangeMenu(hMenu, NULL, "A&bout Output...", ID_ABOUT,
        MF_APPEND | MF_STRING);
    break;
```

The **CreateSolidBrush** functions create the solid brushes to be used to fill the rectangle, the ellipse, and the circle. The **CreatePen** functions create the dotted and dashed lines used to draw borders.

4.6.3 Add the WM_PAINT Case

Add the following case statement to the window function:

```
case WM_PAINT:

    hDC = BeginPaint(hWnd, &ps);

    TextOut(hDC, 1, 1, lpText, _lstrlen(lpText));

    SetRect(&rcTextBox, 1, 10, 161, 44);

    DrawText(hDC, lpText, _lstrlen(lpText),
        &rcTextBox, DT_LEFT | DT_WORDBREAK);

    hOldBrush = SelectObject(hDC, hRedBrush);
    Rectangle (hDC, 10, 50, 60, 80);

    SelectObject(hDC, hGreenBrush);
    Ellipse (hDC, 160, 50, 210, 80);

    SelectObject(hDC, hBlueBrush);
    Pie (hDC, 310, 50, 360, 100, 360, 50, 360, 100);

    SelectObject(hDC, hOldBrush);

    hOldPen = SelectObject(hDC, hDashPen);

    MoveTo(hDC, 10, 110);

    LineTo(hDC, 360, 110);

    SelectObject(hDC, hDotPen);

    Arc(hDC, 10, 90, 360, 130, 10, 110, 360, 110);

    SelectObject(hDC, hOldPen);

    EndPaint(hWnd,  &ps);
    break;
```

4.6.4 Modify the WM_DESTROY Case

You need to delete the drawing tools created for Output's window before terminating the application. You can do this by using the **DeleteObject** function to delete the various pens and brushes in the WM_DESTROY case. Modify the WM_DESTROY case so that it looks like this:

```
case WM_DESTROY:

    DeleteObject(hRedBrush);
    DeleteObject(hGreenBrush);
    DeleteObject(hBlueBrush);
    DeleteObject(hDashPen);
    DeleteObject(hDotPen);
    PostQuitMessage(0);
    break;
```

You need one **DeleteObject** function call for each object to be deleted.

4.6.5 Add the _lstrlen Function

In Output, all string variables have the **LPSTR** type, meaning their addresses are passed as 32-bit values; so to compute a string's length you need to add the locally defined _lstrlen function (described in Section 4.5, "Computing a String's Length") to the source file. Place the following function declaration before the window function:

```
int _lstrlen(lpString)
LPSTR lpString;
{
    int i;
    for (i = 0; *lpString++; ++i);
    return (i);
}
```

4.6.6 Compile and Link

No changes are required to the **make** file to recompile and link the Output application. After compiling and linking Output, start Windows and the application. The application should look like Figure 4.1:

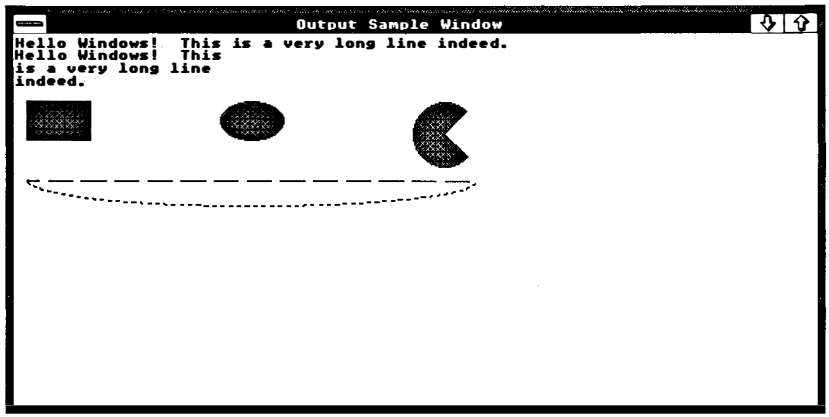


Figure 4.1 Output Window

You can use the WM_PAINT case of this application to experiment with a variety of GDI functions. For information about other GDI output functions, see the *Microsoft Windows Programmer's Reference.*

Chapter 5
Keyboard and Mouse Input

5.1 Introduction

This chapter describes the input messages and explains how to use them in your applications. Windows supplies input messages in response to user input through the keyboard and mouse, and in response to timer input.

5.2 Input Types

Windows provides the following types of input messages:

Message	Description
Keyboard	User input through the keyboard
Character	Keyboard input translated into character codes
Mouse	User input through the mouse
Menu	User input through a window's menus and the mouse
Scroll-bar	User input through a window's scroll bars and the mouse
Timer	Input through the system timer

The keyboard, mouse, and timer input messages correspond directly to hardware input. Windows passes these messages to the application through the application queue. The character, menu, and scroll-bar input are created in response to mouse and keyboard actions in the non-client area of a window, or are the result of translated keyboard messages. Windows typically sends these messages directly to the window function.

5.2.1 Message Format

Input messages have two formats. Messages that Windows places in the application queue have the form of an **MSG** structure. This structure has fields that identify the message and that contain information about the message. The **GetMessage** function in your application's message loop retrieves this structure, and the **DispatchMessage** function takes it as an argument.

The second format of an input message is how the window function receives the message: as four arguments corresponding to the window function's hWnd, message, wParam, and lParam parameters. These parameters receive the same values as given in the input message's **MSG** structure. The only difference is that the **MSG** structure includes a field to specify the location of the mouse when the message was generated and

the system time when the message was generated. The window function does not receive this information.

5.2.2 Keyboard Input

Much of an application's user input comes from the keyboard. Windows sends keyboard input to an application when the user presses or releases a key. The following is a list of the keyboard messages and the events that cause them:

Message	Description
WM_KEYDOWN	User presses a key.
WM_KEYUP	User releases a key.
WM_SYSKEYDOWN	User presses a system key.
WM_SYSKEYUP	User releases a system key.

The wParam parameter of each key specifies the virtual keycode of the given key. A virtual keycode is a device-independent value for a specific keyboard key. Windows uses virtual keycodes to provide consistent keyboard input no matter what computer your application is running on. The lParam parameter contains the keyboard's actual scan code for the key, as well as additional information about the keyboard, such as the state of the SHIFT key and whether the current key was previously up or down.

Windows generates system-key messages, WM_SYSKEYUP and WM_SYSKEYDOWN, for the system keys. These are special keys, such as the ALT and F10 keys, that belong to the Windows user interface and cannot be used by an application in any other way.

An application receives keyboard messages only when it has the "input focus." Your application receives the input focus when it is the active application; that is, when the user has selected your application's window. You can also use the **SetFocus** function to explicitly set the input focus for a given window, and the **GetFocus** function to determine which window has the focus.

5.2.3 Character Input

Applications that read character input from the keyboard need to use the **TranslateMessage** function in their message loops. **TranslateMessage** translates a keyboard-input message into a corresponding ANSI-character message, WM_CHAR or WM_SYSCHAR. These messages contain the ANSI character codes for the given key in the wParam parameter. The lParam parameter is identical to lParam in the keyboard-input message.

5.2.4 Mouse Input

User input can also come from the mouse. Windows sends mouse messages to the application when the user moves the mouse cursor (pointer) into and through a window or presses or releases a mouse cursor while the mouse button is in the window. The following is a list of the mouse messages and the events that cause them:

Message	Description
WM_MOUSEMOVE	User moves the mouse cursor into or through the window.
WM_LBUTTONDOWN	User presses the left button.
WM_LBUTTONUP	User releases the left button.
WM_LBUTTONDBLCLK	User presses, releases, and presses again the left button within the system's defined double-click time.
WM_MBUTTONDOWN	User presses the middle button.
WM_MBUTTONUP	User releases the middle button.
WM_MBUTTONDBLCLK	User presses, releases, and presses again the middle button within the system's defined double-click time.
WM_RBUTTONDOWN	User presses the right button.
WM_RBUTTONUP	User releases the right button.
WM_RBUTTONDBLCLK	User presses, releases, and presses again the right button within the system's defined double-click time.

The wParam parameter of each button includes a bitmask specifying the current state of the keyboard and mouse buttons, such as whether the mouse buttons, SHIFT key, and CONTROL key are down. The lParam parameter contains the the *x*- and *y*-coordinates of the mouse cursor.

Windows sends mouse messages to a window only if the mouse cursor is in the window or if you have captured mouse input by using the **SetCapture** function. The **SetCapture** function directs Windows to send all mouse input, regardless of where the mouse cursor is, to the specified window. Applications typically use this function to take control of it when carrying out some critical operation with the mouse, such as selecting something in the client area. Capturing the mouse prevents other applications from taking control of it before the operation is completed.

Since the mouse is a shared resource, it is important to release the captured mouse as soon as you have finished the operation. You release the mouse by using the **ReleaseCapture** function. To determine which window has the captured mouse, if any, use the **GetCapture** function.

Windows sends double-click messages to a window function only if the corresponding window class has the CS_DBLCLKS style. You must set this style while registering the window class. A double-click message is always the third message in a four-message series. The first two messages are the first button press and release. The second button press is replaced with the double-click message. The last message is the second release. Remember that a double-click message only occurs if the first and second press occur within the system's defined double-click time. You can retrieve the current double-click time by using the **GetDoubleClickTime** function. You can set it by using the **SetDoubleClickTime** function, but be aware that this sets the double-click time for all applications, not just your own.

5.2.5 Timer Input

Windows sends timer input to your application whenever the set time elapses for a timer. To receive timer input, you must set a timer by using the **SetTimer** function. Timer input is received in two ways: as a WM_TIMER message through the system queue, or through a callback function that you specify when you call the **SetTimer** function. The following example shows how to set timer input, using a WM_TIMER message, for a five-second interval:

```
idTimer = SetTimer(hWnd, NULL, 5000, (FARPROC) NULL);
```

This example sets a timer interval of 5000 milliseconds. This means that the timer will generate input every five seconds. The last argument is NULL, meaning that there is no callback function for the timer input, so Windows sends the timer input through the application queue.

The **SetTimer** function returns a unique integer that identifies the timer. You can use this same timer ID to turn the timer off by using it in the **KillTimer** function.

5.2.6 Scroll-Bar Input

Windows sends a scroll-bar input message, either WM_HSCROLL or WM_VSCROLL, to a window function when the user clicks with the mouse cursor in a scroll bar. Applications use the scroll-bar messages to direct scrolling within the window. Applications that display text or other data that does not all fit in the client area usually provide some form of scrolling. Scroll bars are an easy way to let the user direct scrolling actions.

To get scroll-bar input, you need to add scroll bars to the window. You can add scroll bars to a window by specifying the WS_HSCROLL and WS_VSCROLL styles when you create the window. These direct the **CreateWindow** function to create horizontal and vertical scroll bars for the window. The following example creates a scroll bar for the given window:

```
hWnd = CreateWindow("Input",                    /* window class      */
    "Input Sample Application",                 /* window name       */
    WS_OVERLAPPEDWINDOW | WS_HSCROLL | WS_VSCROLL,
    CW_USEDEFAULT,                              /* x position        */
    CW_USEDEFAULT,                              /* y position        */
    CW_USEDEFAULT,                              /* width             */
    CW_USEDEFAULT,                              /* height            */
    NULL,                                       /* parent handle     */
    NULL,                                       /* menu or child ID  */
    hInstance,                                  /* instance          */
    NULL);                                      /* additional info   */
```

Windows displays the scroll bars when it displays the window. It automatically maintains the scroll bars and sends scroll-bar messages to the window function when the user clicks.

When Windows sends a scroll-bar message, it sets the wParam parameter of the message to a value that indicates the type of scrolling request made. For example, if the user clicks the top arrow of a vertical scroll bar, Windows sets the wParam parameter to the value, SB_LINEUP. The following list shows the various values for the wParam parameter and how they are generated:

Scroll Type	Description
SB_LINEUP	User clicks the upper or left arrow.
SB_LINEDOWN	User clicks the lower or right arrow.
SB_PAGEUP	User clicks between the scroll box and the upper or left arrow.
SB_PAGEDOWN	User clicks between the scroll box and the lower or right arrow.
SB_THUMBPOSITION	User releases the mouse button when the mouse cursor is in the scroll box, typically after dragging the box.
SB_THUMBTRACK	User drags the scroll box with the mouse.

5.2.7 Menu Input

Windows sends a menu-input message, either WM_SYSCOMMAND or WM_COMMAND, to a window function whenever the user chooses a command, such as the About command in the system menu. Since menu input is often the primary source of input for an application and its processing can be complex, it is described in detail in Chapter 8, "Menus."

5.3 Displaying Formatted Output

Although you cannot use the C run-time **printf** function to display formatted output, you can use the **sprintf** function to copy a formatted string to a buffer and pass the buffer address as an argument to the **TextOut** function. In small-model applications, such as the sample applications described in this guide, you need to be careful when using the **sprintf** function that you ensure that the buffer you specify is defined as a global or local variable (is within the application's data segment or stack). The following example shows how to display formatted output:

```
char MouseText[40];
    .
    .
    .

sprintf(MouseText, "WM_MOUSEMOVE: %x, %d, %d", wParam,
    LOWORD(lParam), HIWORD(lParam));
TextOut(hDC, 10, 10, MouseText, strlen(MouseText));
```

This example copies the formatted string to the MouseText array. The array is declared a local variable so that it can be passed to the **sprintf** function. It can also be passed to the C run-time **strlen** function, which is used in the **TextOut** function to compute the string length.

5.4 A Sample Application: Input

This sample application illustrates how to process input messages from the keyboard, mouse, timer, and scroll bars. The "Input" application displays the current or most recent state of each of these input mechanisms. To create the Input application, copy and rename the source files of the Generic application, then make the following modifications:

1. Add new variables.
2. Set the window-class style.

3. Modify the **CreateWindow** function.
4. Modify the WM_CREATE and WM_DESTROY cases.
5. Add the WM_KEYUP and WM_KEYDOWN cases.
6. Add the WM_CHAR case.
7. Add the WM_MOUSEMOVE case.
8. Add the WM_LBUTTONUP and WM_RBUTTONUP cases.
9. Add the WM_LBUTTONDBLCLK case.
10. Add the WM_TIMER case.
11. Add the WM_HSCROLL and WM_VSCROLL cases.
12. Add the WM_PAINT case.
13. Compile and link.

Although Windows does not require a pointing device, this sample assumes that you have a mouse or other pointing device. If you do not have a mouse, the application will not receive mouse-input messages.

5.4.1 Add New Variables

You need several new global variables. Declare the following variables at the beginning of the C-language source file:

```
char MouseText[40];                /* mouse state          */
char ButtonText[40];               /* mouse-button state   */
char KeyboardText[40];             /* keyboard state       */
char CharacterText[40];            /* latest character     */
char ScrollText[40];               /* scroll status        */
char TimerText[40];                /* timer state          */
int idTimer;                       /* timer ID             */
int nTimerCount = 0;               /* current timer count  */
```

These character arrays hold strings that describe the current state of the keyboard, mouse, and timer.

You also need some local variables for the window function. Declare the following variables at the beginning of the window function:

```
HDC hDC;                           /* display-context variable */
PAINTSTRUCT ps;                    /* paint structure          */
```

5.4.2 Set the Window-Class Style

You need to set the window-class style to CS_DBLCLKS to enable double-click processing. In the initialization function, find this statement:

```
pWndClass->style = NULL;
```

Change it to the following:

```
pWndClass->style = CS_DBLCLKS;          /* double-click messages */
```

This enables double-click processing for windows that belong to this class.

5.4.3 Modify the CreateWindow Function

You need modify the call to the **CreateWindow** function in order to create a window that has vertical and horizontal scroll bars. Change the **CreateWindow** function call in the **WinMain** function so that it looks like this:

```
hWnd = CreateWindow("Input",
    "Input Sample Window",
    WS_OVERLAPPEDWINDOW |
    WS_HSCROLL | WS_VSCROLL,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL);
```

5.4.4 Modify the WM_CREATE and WM_DESTROY Cases

You need to set a timer by using the **SetTimer** function. You can do this in the WM_CREATE case. Add the following statement:

```
idTimer =  SetTimer(hWnd, NULL, 5000, (FARPROC) NULL);
```

You also need to stop the timer before terminating the application. You can do this in the WM_DESTROY case. Add the following statement:

```
KillTimer(hWnd, idTimer);
```

5.4.5 Add the WM_KEYUP and WM_KEYDOWN Cases

You need to add the WM_KEYUP and WM_KEYDOWN cases to process key presses. Add the following statements to the window function:

```
case WM_KEYDOWN:
    sprintf(KeyboardText, "WM_KEYDOWN: %x, %x, %x     ",
        wParam, LOWORD(lParam), HIWORD(lParam));
    InvalidateRect(hWnd, NULL, FALSE);
    break;

case WM_KEYUP:
    sprintf(KeyboardText, "WM_KEYUP: %x, %x, %x     ",
        wParam, LOWORD(lParam), HIWORD(lParam));
    InvalidateRect(hWnd, NULL, FALSE);
    break;
```

5.4.6 Add the WM_CHAR Case

You need to add a WM_CHAR case to process ANSI-character input. Add the following statements to the window function:

```
case WM_CHAR:
    sprintf(CharacterText, "WM_CHAR: %c, %x, %x     ",
        wParam, LOWORD(lParam), HIWORD(lParam));
    InvalidateRect(hWnd, NULL, FALSE);
    break;
```

5.4.7 Add the WM_MOUSEMOVE Case

You need to add a WM_MOUSEMOVE case to process mouse-motion messages. Add the following statements to the window function:

```
case WM_MOUSEMOVE:
    sprintf(MouseText, "WM_MOUSEMOVE: %x, %d, %d     ",
        wParam, LOWORD(lParam), HIWORD(lParam));
    InvalidateRect(hWnd, NULL, FALSE);
    break;
```

5.4.8 Add the WM_LBUTTONUP and WM_LBUTTONDOWN Cases

You need to add the WM_LBUTTONUP and WM_LBUTTONDOWN cases to process mouse-button input messages. Add the following statements to the window function:

```
case WM_LBUTTONDOWN:
    sprintf(ButtonText, "WM_LBUTTONDOWN: %x, %d, %d     ",
        wParam, LOWORD(lParam), HIWORD(lParam));
```

```
    InvalidateRect(hWnd, NULL, FALSE);
    break;

case WM_LBUTTONUP:
    sprintf(ButtonText, "WM_LBUTTONUP: %x, %d, %d     ",
        wParam, LOWORD(lParam), HIWORD(lParam));
    InvalidateRect(hWnd, NULL, FALSE);
    break;
```

5.4.9 Add the WM_LBUTTONDBLCLK Case

You need to add a WM_LBUTTONDBLCLK case to process mouse-button input messages. Add the following statements to the window function:

```
case WM_LBUTTONDBLCLK:
    sprintf(ButtonText, "WM_LBUTTONDBLCLK: %x, %d, %d     ",
        wParam, LOWORD(lParam), HIWORD(lParam));
    InvalidateRect(hWnd, NULL, FALSE);
    break;
```

5.4.10 Add the WM_TIMER Case

You need to add a WM_TIMER case to process timer messages. Add the following statements to the window function:

```
case WM_TIMER:
    sprintf(TimerText, "WM_TIMER: %d seconds     ",
        nTimerCount += 5);
    InvalidateRect(hWnd, NULL, FALSE);
    break;
```

5.4.11 Add the WM_HSCROLL and WM_VSCROLL Cases

You need to add the WM_HSCROLL and WM_VSCROLL cases to process scroll-bar messages. Add the following statements to the window function:

```
case WM_HSCROLL:
case WM_VSCROLL:
    sprintf(ScrollText, "%s: %s, %x, %x     ",
        (message == WM_HSCROLL) ? "WM_HSCROLL" : "WM_VSCROLL",
        (wParam == SB_LINEUP) ? "SB_LINEUP" :
        (wParam == SB_LINEDOWN) ? "SB_LINEDOWN" :
        (wParam == SB_PAGEUP) ? "SB_PAGEUP" :
        (wParam == SB_PAGEDOWN) ? "SB_PAGEDOWN" :
        (wParam == SB_THUMBPOSITION) ? "SB_THUMBPOSITION" :
        (wParam == SB_THUMBTRACK) ? "SB_THUMBTRACK" :
        (wParam == SB_ENDSCROLL) ? "SB_ENDSCROLL" : "unknown",
        LOWORD(lParam), HIWORD(lParam));
    InvalidateRect(hWnd, NULL, FALSE);
    break;
```

5.4.12 Add the WM_PAINT Case

You need to display the current mouse, keyboard, and timer states. The most convenient way to do this is to use the WM_PAINT message to display the states. Add the following statements to the window function:

```
case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);

    TextOut(hDC, 20, 20, MouseText, strlen(MouseText));
    TextOut(hDC, 20, 40, ButtonText, strlen(ButtonText));
    TextOut(hDC, 20, 60, KeyboardText, strlen(KeyboardText));
    TextOut(hDC, 20, 80, CharacterText, strlen(CharacterText));
    TextOut(hDC, 20, 100, TimerText, strlen(TimerText));
    TextOut(hDC, 20, 120, ScrollText, strlen(ScrollText));

    EndPaint(hWnd, &ps);
    break;
```

5.4.13 Compile and Link

You can compile and link the Input application without changing the **make** file. Once the application is compiled, start Windows and then the Input application. To test the application, press keys on the keyboard, click the mouse button, and move the mouse. The application should look like Figure 5.1:

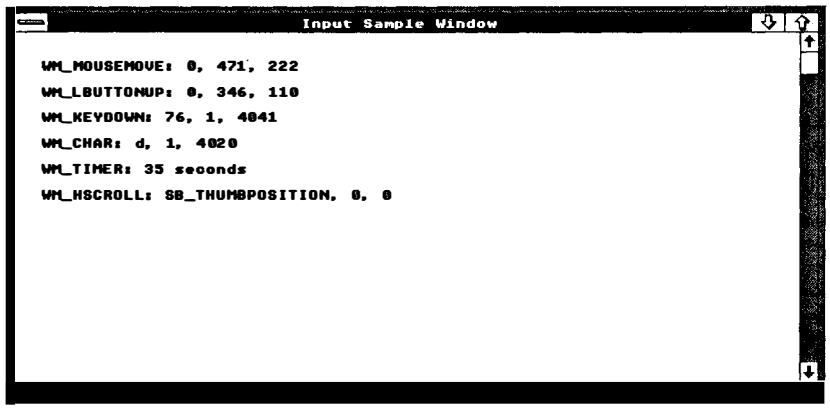


Figure 5.1 Input Window

Chapter 6
Icons

6.1 What Are Icons?

An icon is a special bitmap that you may use to graphically represent a window or other object associated with your application. Icons are typically used to represent an application when its main window is minimized. For example, Microsoft Paint uses an icon that looks like a painter's palette to represent its minimized window. Icons are also used in message and dialog boxes.

An icon is not just a bitmap. In fact, it is a composite of two bitmaps, which when displayed provide special effects, such as a transparent background. When you load an icon, Windows may adjust its size to match the resolution of the particular display (Windows does not do this for bitmaps). This means that you can design and use device-independent icons and be guaranteed that these icons will look reasonable no matter what display they appear on.

6.2 Class Icons

A class icon is an icon that is used for a particular class of windows each time a window in that class is minimized. You set a class icon by assigning an icon handle to the **hIcon** field of the window-class structure before registering the class. Once the class icon is set, any window you create using that class will display the class icon when it is minimized. The following statement shows how to set a class icon:

```
pWndClass->hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

The **LoadIcon** function returns a handle to the built-in application icon identified by IDI_APPLICATION. If you minimize a window that has this class, you will see a white rectangle with a black border. This is the built-in application icon.

Windows provides several built-in icons. Choices include the exclamation point, question mark, hand, and asterisk, as well as the application icon (for examples of these icons, see the *Microsoft Windows Application Style Guide*). You can use any of these icons in your applications. Windows uses most of them in message boxes to represent notes, cautions, warnings, and errors.

To use a built-in icon, you retrieve a handle to it by using the **LoadIcon** function. The first argument to the function must be NULL, and the

second must identify the icon you want. For example, if you want to use the hand icon, you use the following function:

```
hHandIcon = LoadIcon(NULL, IDI_HAND);
```

The NULL argument indicates that a built-in icon is requested.

6.3 Creating Icons

Creating an icon requires three simple steps: create the icon by using the Windows 2.0 Icon Editor, add an **ICON** statement to your resource script file, and load the icon, when needed, by using the **LoadIcon** function. Figure 6.1 shows the first step—an icon being edited in Icon Editor:

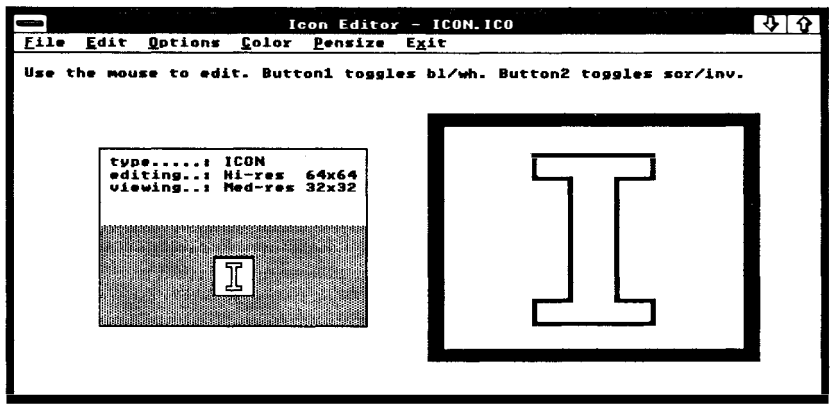


Figure 6.1 Icon Editor with an Icon

Follow the directions given in *Microsoft Windows Programming Tools* for creating an icon, then save the icon in a file. The recommended file extension for an icon is *.ico*.

Next, add an **ICON** statement to your resource script file. For example, the following statement adds the icon named "icon" to your application's resources:

```
icon ICON icon.ico
```

The filename, *icon.ico*, specifies the file containing the icon. When the resource script file is compiled, the icon will be copied from the specified file into your application's resources.

Finally, load the icon from your resources by using the **LoadIcon** function. You need to specify the name of the icon and the application's instance handle:

```
pWndClass->hIcon = LoadIcon( hInstance, (LPSTR) "icon");
```

In this example, the loaded icon is used as the class icon.

6.4 Creating Your Own Icons

You can create your own icon when a window is minimized by setting the class icon to NULL and creating the icon when the window function receives a WM_PAINT message. Windows lets applications paint within the client area of an iconic window, creating a dynamic icon such as the one in the Clock application. (The Clock application continues to show the time even when it has been minimized.)

To create an icon, you must set the class icon to NULL, then add a WM_PAINT case to your window function to draw within the icon's client area. The first step, setting the class icon to NULL, must be done before you register the window class. Use the following statement:

```
pWndClass->hIcon = NULL;
```

This step is required because it signals Windows to continue sending WM_PAINT messages, as necessary, to the window function even though the window has been minimized.

You can draw within the icon's client area by processing the WM_PAINT message. Add the following case statement to the window function:

```
PAINTSTRUCT ps;
    .
    .
    .
case WM_PAINT:
    if (IsIconic(hWnd)) {
        hDC = BeginPaint(hWnd, &ps);
        .
        /* Place output functions here */
        .
        EndPaint(hWnd, &ps);
    }
    break;
```

Applications need to determine whether the window is iconic, since what they paint in the icon may be different from what they paint in the open window. The **IsIconic** function returns TRUE if the window is iconic.

The **BeginPaint** function returns a handle to the display context of the icon's client area. **BeginPaint** takes the window handle, hWnd, and a long pointer to the paint structure, ps. **BeginPaint** fills the paint structure with information about the area to be painted. As with any painting operation, after each call to **BeginPaint**, the **EndPaint** function is required. **EndPaint** releases any resources that **BeginPaint** retrieved and signals the end of the application's repainting of the client area.

You can retrieve the size of the icon's client area by using the **rcPaint** field of the paint structure. For example, if you want to draw an ellipse that fills the icon, you can use the following statement:

```
Ellipse(hDC, ps.rcPaint.left, ps.rcPaint.top,
        ps.rcPaint.right, ps.rcPaint.bottom);
```

You can use any GDI output functions to draw the icon, including the **TextOut** function. The only limitation is the size of the icon, which varies from display to display, so make sure that your painting does not depend on a specific icon size.

6.5 Using an Icon in a Dialog Box

You can place icons in dialog boxes by using the **ICON** control statement in the **DIALOG** statement. You have already seen an example of a **DIALOG** statement in the About dialog box described with the Generic application. The **DIALOG** statement for that box looks like this:

```
AboutBox DIALOG 22, 17, 144, 75
STYLE WS_POPUP | WS_DLGFRAME
BEGIN
    CTEXT "Microsoft Windows"               -1, 37,  5,  68, 8
    CTEXT "Generic Application"             -1,  0, 14, 144, 8
    CTEXT "Version 1.0"                     -1, 38, 34,  64, 8
    DEFPUSHBUTTON "OK"          IDOK, 53, 59, 32, 14, WS_GROUP
END
```

You can add an icon to the dialog box by inserting the following **ICON** statement immediately after the **DEFPUSHBUTTON** statement:

```
ICON "icon", -1, 25, 14, 16, 21
```

The name "icon" identifies the icon to be used. The icon must be defined in an **ICON** statement elsewhere within the resource script file. For example, adding the following statement to the resource file satisfies the control statement:

```
icon ICON icon.ico
```

When an icon is added to a dialog box, it is treated like any other control. It must have a control ID, a position for its upper-left corner, a width, and a height.

6.6 A Sample Application: Icon

This sample application shows how to incorporate icons in your applications, in particular, how to do the following:

- Use a custom icon as the class icon.
- Use an icon in the About dialog box.

To create the Icon application, copy and rename the source files of the Generic application, then make the following modifications:

1. Add an **ICON** statement to the resource script file.
2. Add an **ICON** control statement to the **DIALOG** statement in the resource script file.
3. Load the custom icon and use it to set the class icon in the initialization function.

This sample assumes that you have created an icon by using Icon Editor and have saved it in the file named *icon.ico*.

6.6.1 Add an ICON Statement

You need to add an ICON statement to your resource script file. Insert the following line at the beginning of the resource script file, immediately after the **#include** statements:

```
icon ICON icon.ico
```

6.6.2 Add an ICON Control Statement

You need to add an **ICON** control statement to the **DIALOG** statement. Insert the following line immediately after the **DEFPUSHBUTTON** statement:

```
ICON "icon", -1, 25, 14, 16, 21
```

6.6.3 Set the Class Icon

You can set the class icon by adding the following statement to the initialization function in the C-language source file:

```
pWndClass->hIcon = LoadIcon(hInstance, "icon");
```

No other changes are required.

6.6.4 Compile and Link

No changes are required to the **make** file to recompile and link the Icon application. When the application is recompiled, start Windows and the Icon application. Now, if you choose the About command, the About dialog box will look like Figure 6.2:

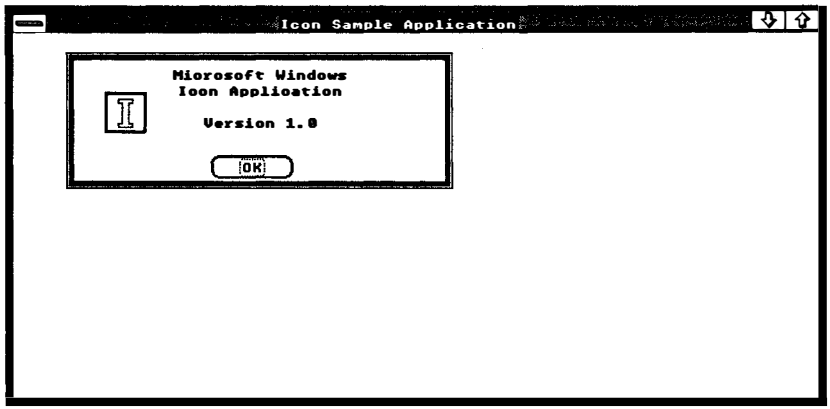


Figure 6.2 The About Dialog Box in Icon

Chapter 7
The Cursor, the Mouse, and the Keyboard

7.1 Introduction

The system cursor is a bitmap that shows the user where actions initiated by the mouse will take place. Most applications use the cursor with either the mouse or the keyboard to let the user make selections, choose commands, and direct other actions within an application. Some of these actions are carried out automatically by Microsoft Windows; others must be carried out by the application. This chapter explains the purpose of the system cursor and shows how to use it in your applications. It also explains how to use the keyboard to carry out functions similar to those of the mouse, and how to use the cursor when a mouse or other pointing device is not available in the system.

7.2 Using the Cursor

Since no one cursor shape can satisfy the needs of all applications, Windows lets you change the shape of the cursor to a shape appropriate to your application and the actions it carries out. Within your application, you can control the shape of the cursor in a window either by setting the "class cursor" or by using the **SetCursor** function to explicitly set the shape when the cursor moves within the client area of the window. The following sections explain these two methods.

7.2.1 Class Cursor

A class cursor defines the shape the cursor will take when it enters the client area of a window belonging to that class. You specify a class cursor by assigning a cursor handle the **hCursor** field of the window-class structure before registering the class. For example, to load the built-in arrow cursor (IDC_ARROW) in your window, you can add the following statement to your initialization function:

```
pWndClass->hCursor = LoadCursor(NULL, IDC_ARROW);
```

For each window created using this class, the built-in arrow cursor will appear when the user moves the cursor into the window.

Windows provides several built-in cursor shapes. These include the arrow, hourglass, I-beam, and cross-hair cursors. An application can use these shapes for its class cursors by using the **LoadCursor** function to retrieve handles to the shapes. To load a built-in cursor, the first argument must

be NULL (indicating that a built-in cursor is requested), and the second argument must specify the cursor to load; for example, the I-beam cursor (IDC_BEAM), which is typically used in windows that let the user view and edit text:

```
pWndClass->hCursor = LoadCursor(NULL, IDC_IBEAM);
```

Built-in cursors can also be loaded for purposes other than the class cursor. For example, the hourglass cursor is a commonly used to indicate a lengthy operation, such as reading or writing to a disk file. As such, it is used only while the lengthy operation is in progress—as described later in this chapter.

7.2.2 Creating Cursors

You can create and use your own cursor shapes by following three simple steps: create the cursor by using the Windows 2.0 Icon Editor, add the cursor to your resources by using the **CURSOR** statement, and load the cursor by using the **LoadCursor** function. Figure 7.1 shows the first step—a cursor being edited in Icon Editor:

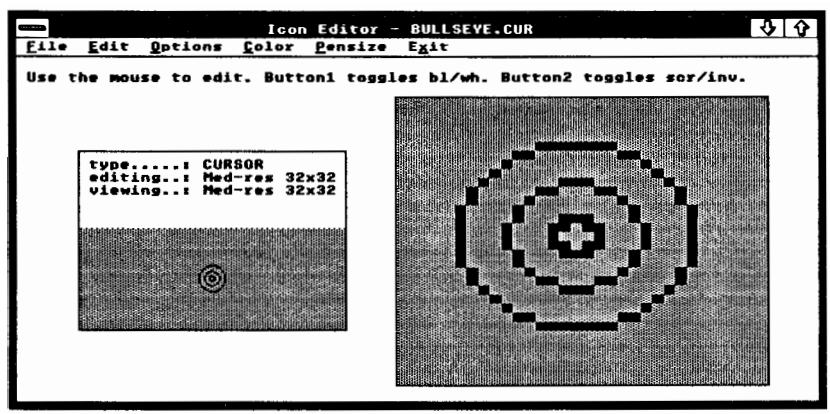


Figure 7.1 Icon Editor and a Cursor

When you have created the cursor, save it in a file by using the *.cur* filename extension. This is the recommended extension for cursor files.

Next, you need to add a **CURSOR** statement to your resource script file. The **CURSOR** statement specifies the file that contains the cursor and the name to be used by the application when loading the cursor:

```
bullseye CURSOR bullseye.cur
```

In this example, the name of the cursor is "bullseye", and the cursor is in the file *bullseye.cur*.

Finally, you need to load the cursor and assign its handle to the **hCursor** field of the window-class structure. Do this before registering the class:

```
pWndClass->hCursor = LoadCursor(hInstance, (LPSTR) "bullseye");
```

The **LoadCursor** function loads the cursor from the application's resources. The instance handle, hInstance, identifies the application's resources and is required. The name "bullseye" identifies the cursor. It is the same name given in the resource script file.

7.2.3 Displaying Your Own Cursor

An application does not have to define a class cursor. Instead, the application can set the **hCursor** field to NULL to indicate no class cursor. If a window has no class cursor, Windows will not automatically change the shape of the cursor when it moves into the client area of the window. This means you will need to display your own cursor.

To change the cursor shape, you need to use the **SetCursor** function to set the shape each time the cursor moves in the client area. Since Windows sends a WM_MOUSEMOVE message to the window on each cursor movement, you can manage the cursor by adding the following statements to the window function:

```
case WM_MOUSEMOVE:
    SetCursor(hMyCursor);
    break;
```

To display a cursor, whether built-in or custom, you still need to load it. In this example, the handle of the loaded cursor has been assigned to the variable hMyCursor.

Note

If you choose to display your own cursor, you must make sure you set the class-cursor field to NULL. Otherwise, Windows will attempt to change the cursor shape even though you do so on each WM_MOUSEMOVE message. This will result in a noticeable flicker as you move the cursor through the window.

7.2.4 Showing the Hourglass on a Lengthy Operation

Whenever your application begins a lengthy operation, such as reading or writing a large block of data to a disk file, you should change the shape of the cursor to the hourglass. This lets users know that a lengthy operation is in progress and that they should wait before attempting to continue their work. Once the operation is complete, you should restore the cursor to its previous shape.

You can change the shape of the cursor by using the following statements:

```
HCURSOR hSaveCursor;
HCURSOR hHourGlass;
     .
     .
     .
hHourGlass = LoadCursor(hInstance, IDC_WAIT);
     .
     .
     .
SetCapture(hWnd);
hSaveCursor = SetCursor(hHourGlass);

/* Lengthy operation */

SetCursor(hSaveCursor);
ReleaseCapture();
     .
     .
     .
```

In this example, the application first captures the mouse input, using the **SetCapture** function. This keeps the user from attempting to use the mouse to carry out work in another application while the lengthy operation is in progress. When the mouse input is captured, Windows directs it to the specified window, regardless of whether the mouse is in that window. The cursor shape is then set by using the **SetCursor** function. The previous shape returned by **SetCursor** is saved so that it can be restored by using **SetCursor** again when the operation is complete. The **ReleaseCapture** function releases the mouse input.

7.3 Using the Mouse

The mouse lets the user move a cursor on the screen and enter simple input through the press of a button. You can use the mouse to carry out many types of tasks, such as choosing commands from a menu, selecting text or graphics, or directing scrolling operations. Windows carries out many of these tasks automatically, but one common task, selection, must be done by the application itself. The following sections explain how to use mouse input to select graphics in a window's client area.

The mouse is just one of many possible system pointing devices. Other pointing devices such as graphics tablets, joysticks, and light pens may operate differently but still provide input identical to that of a mouse. The following examples can be used with these devices as well. Remember that when a pointing device is present, Windows automatically controls the position and shape of the cursor as the user moves the pointing device.

7.3.1 Starting a Graphics Selection

A simple approach to selecting graphics is to determine a rectangle within a window's client area and invert the border of the rectangle to show that it has been selected. You can use the messages WM_LBUTTONDOWN, WM_LBUTTONUP, and WM_MOUSEMOVE to create the rectangle. This lets the user create the selection by choosing a point, pressing the left button, and dragging to another point before releasing. While the user drags the mouse, the application can provide instant feedback by inverting the border of the rectangle described by the starting and current points.

For this method, you start the selection when you receive the message WM_LBUTTONDOWN. You need to do three things: capture the mouse input, save the starting (original) point, and save the current point:

```
BOOL bTrack = FALSE;
int OrgX = 0, OrgY = 0;
int PrevX = 0, PrevY = 0;
int X = 0, Y = 0;
    .
    .
    .
case WM_LBUTTONDOWN:
    bTrack = TRUE;
    OrgX = LOWORD(lParam);
    OrgY = HIWORD(lParam);
    PrevX = LOWORD(lParam);
    PrevY = HIWORD(lParam);
    SetCapture(hWnd);
    break;
```

When the application receives the WM_LBUTTONDOWN message, the bTrack variable is set to TRUE to indicate that a selection is in progress. As with any mouse message, the lParam parameter contains the current *x*- and *y*-coordinates of the mouse in the low- and high-order words, respectively. These are saved as the original *x* and *y* values, OrgX and OrgY, as well as the previous values, PrevX and PrevY. The PrevX and PrevY variables will be updated immediately on the next WM_MOUSE-MOVE message. The OrgX and OrgY variables remain unchanged and will be used to determine a corner of the bitmap to be copied. The **SetCapture** function directs all subsequent mouse input to the window even if the cursor moves outside of the window. This is to ensure that the selection process continues uninterrupted. The variables bTrack, OrgX, OrgY, PrevX, and PrevY must be global variables.

If there is any previous selection, it should be cleared before starting the new selection. Clearing the selection means restoring the inverted screen to its previous state. You can restore the inverted screen by adding the following statements to the beginning of the WM_LBUTTONDOWN case:

```
if (OrgX != X || OrgY != Y) {   /* Clears previous box */
   hDC = GetDC(hWnd);
   SetROP2(hDC, R2_NOT);
   MoveTo(hDC, OrgX, OrgY);
   LineTo(hDC, OrgX, Y);
   LineTo(hDC, X, Y);
   LineTo(hDC, X, OrgY);
   LineTo(hDC, OrgX, OrgY);
   ReleaseDC(hWnd, hDC);
}
```

In some applications, you may want to be able to extend an existing selection. One way to do this is to have the user hold the SHIFT key when creating a selection. Since the wParam parameter contains a flag that specifies whether the SHIFT key is being pressed, it is easy to check for this and to extend the selection, as necessary. In this case, extending a selection means preserving its previous OrgX and OrgY values when you start it. To do this, change the WM_LBUTTONDOWN case so it looks like this:

```
case WM_LBUTTONDOWN:
    bTrack = TRUE;

    if (OrgX != X || OrgY != Y) {   /* Clears previous box */
    hDC = GetDC(hWnd);
    SetROP2(hDC, R2_NOT);
    MoveTo(hDC, OrgX, OrgY);
    LineTo(hDC, OrgX, Y);
    LineTo(hDC, X, Y);
    LineTo(hDC, X, OrgY);
    LineTo(hDC, OrgX, OrgY);
    ReleaseDC(hWnd, hDC);
}
    PrevX = LOWORD(lParam);
    PrevY = HIWORD(lParam);
    if (!(wParam & MK_SHIFT)) { /* If shift key is not pressed */
       OrgX = LOWORD(lParam);
       OrgY = HIWORD(lParam);
}
else {     /* Shift key is pressed, update the current box */
    hDC = GetDC(hWnd);
    SetROP2(hDC, R2_NOT);
    MoveTo(hDC, OrgX, OrgY);
    LineTo(hDC, OrgX, PrevY);
    LineTo(hDC, PrevX, PrevY);
    LineTo(hDC, PrevX, OrgY);
    LineTo(hDC, OrgX, OrgY);
    ReleaseDC(hWnd, hDC);
}

SetCapture(hWnd);
break;
```

7.3.2 Showing the Selection

As the user makes the selection, you need to provide feedback about his or her progress. You can do this by drawing a border around the rectangle by using the **LineTo** function on each new WM_MOUSEMOVE message. To prevent losing information already on the display, you need to draw a line that inverts the screen rather than drawing over it. You can do this by using the **SetROP2** function to set the binary raster mode to R2_NOT. The following statements perform this function:

```
case WM_MOUSEMOVE:
    if (bTrack) {
       hDC = GetDC(hWnd);
       SetROP2(hDC, R2_NOT);   /* Erases the previous box */
       MoveTo(hDC, OrgX, OrgY);
       LineTo(hDC, OrgX, PrevY);
       LineTo(hDC, PrevX, PrevY);
       LineTo(hDC, PrevX, OrgY);
       LineTo(hDC, OrgX, OrgY);

       PrevX = LOWORD(lParam);
       PrevY = HIWORD(lParam);
       MoveTo(hDC, OrgX, OrgY);       /* Draws the new box */
       LineTo(hDC, OrgX, PrevY);
       LineTo(hDC, PrevX, PrevY);
       LineTo(hDC, PrevX, OrgY);
       LineTo(hDC, OrgX, OrgY);
       ReleaseDC(hWnd, hDC);
}
break;
```

A WM_MOUSEMOVE message is processed only if bTrack is TRUE (that is, if a selection is in progress). The purpose of the WM_MOUSEMOVE processing is to remove the border around the previous rectangle and draw a new border around the rectangle described by the current and original position. Since the border is actually the inverse of what was originally on the display, inverting again restores it completely. The first four **LineTo** functions remove the previous border. The next four draw a new border. Before drawing the new border, the PrevX and PrevY values are updated by assigning them the current values contained in the lParam parameter.

7.3.3 Ending the Selection

Finally, when the user releases the left button, you need to save the final point and signal the end of the selection process. The following statements complete the selection:

```
case WM_LBUTTONUP:
    bTrack = FALSE;          /* Ignores mouse input          */
    ReleaseCapture();        /* Releases hold on mouse input */

    X = LOWORD(lParam);      /* Saves the current value      */
    Y = HIWORD(lParam);
    break;
```

When the application receives a WM_LBUTTONUP message, it immediately sets the value of bTrack to FALSE to indicate that selection processing has been completed. It also releases the mouse capture by using the **ReleaseCapture** function. It then saves the current mouse position in the variables, X and Y.

For some applications, you may want to check the final mouse position to make sure it represents a point to the lower right of the original point. This is the way most rectangles are described—by their upper-left and lower-right corners.

The **ReleaseCapture** function is required since a corresponding **SetCapture** function was called. In general, you should release the mouse immediately after the mouse capture is no longer needed.

7.4 Using the Cursor with the Keyboard

Windows does not require a pointing device, so many applications that would otherwise use the mouse for input must provide the user with a way to duplicate these actions with the keyboard. Applications that use the cursor to track keyboard motion can use the **SetCursorPos**, **SetCursor**, **GetCursorPos**, **ClipCursor**, and **ShowCursor** functions to display and move the cursor.

7.4.1 Using the Keyboard to Move the Cursor

You can use the **SetCursorPos** function to move the cursor directly from your application. This function is typically used to let the user move the cursor by using the keyboard.

To move the cursor, use the WM_KEYDOWN message and filter for the virtual key values of the DIRECTION keys: VK_LEFT, VK_RIGHT, VK_UP, and VK_DOWN. On each each keystroke, you can update the position of the cursor:

```
POINT ptCursor;
    .
    .
    .
case WM_KEYDOWN:
    if (wParam != VK_LEFT || wParam != VK_RIGHT ||
        wParam != VK_UP || wParam != VK_DOWN )
        break;

    GetCursorPos(&ptCursor);
    ScreenToClient(hWnd, &ptCursor);

    switch (wParam) {
```

```
        case VK_LEFT:
            ptCursor.x -= 1;
            break;
        case VK_RIGHT:
            ptCursor.x += 1;
            break;
        case VK_UP:
            ptCursor.y -= 1;
            break;
        case VK_DOWN:
            ptCursor.y += 1;
            break;
    }

    ClientToScreen(hWnd, &ptCursor);
    SetCursorPos(&ptCursor);
    break;
```

If the mouse is also available, you may want to retrieve the current cursor by using the **GetCursorPos** function. Since the user could potentially move the cursor with the mouse at any time, there is no guarantee that the position values you saved on the last keystroke are correct. The example shows how to retrieve the cursor position and convert the coordinates to client coordinates.

The **SetCursorPos** function moves the cursor to the desired location. Notice that the **SetCursorPos** function requires screen coordinates rather than client coordinates. This means that you need to convert the coordinates before calling the function, by using the **ClientToScreen** function. In this example, the cursor position is saved in client coordinates for two reasons: mouse messages give the mouse position in client coordinates, and client coordinates do not need to be updated if the window moves. In other words, it is convenient to use client coordinates because the system uses them and because it means less work for the application.

You should also check the cursor motion so that it remains within the client area. A simple way to check this is to retrieve the current size of the client area by using the **GetClientRect** function:

```
RECT Rect;
    .
    .
    .
GetClientRect(hWnd, &Rect);
    break;
```

You can then check the current cursor position before setting it, and, if necessary, adjust it:

```
if (ptCursor.x >= Rect.right)
    ptCursor.x = Rect.right - 1;
else if (ptCursor.x < Rect.left)
    ptCursor.x = Rect.left;
if (ptCursor.y >= Rect.bottom)
    ptCursor.y = Rect.bottom - 1;
```

```
else if (ptCursor.y < Rect.top)
    ptCursor.y = Rect.top;
```

Another enhancement you might make is to allow for accelerated cursor motion. Advancing the cursor one unit for each keystroke can be frustrating for users if they need to move to the other side of the screen. You can accelerate the cursor motion by increasing the number of units the cursor advances when the user holds down a key. When the user holds down a key, Windows sends multiple WM_KEYDOWN messages without matching WM_KEYUP messages. To accelerate the cursor, you simply increase the number of units to advance on each WM_KEYDOWN message. The following statements show how to do this:

```
int repeat = 1;
    .
    .
    .
repeat++;                /* Increases the repeat rate */

switch (wParam) {

    case VK_LEFT:
        ptCursor.x -= repeat;
        break;

    case VK_RIGHT:
        ptCursor.x += repeat;
        break;

    case VK_UP:
        ptCursor.y -= repeat;
        break;

    case VK_DOWN:
        ptCursor.y += repeat;
        break;

    default:
        return (NULL);
}
```

You need to restore the initial value of the repeat variable when the user releases the key. You can do this by using the WM_KEYUP message. The following statements show how to do this:

```
case WM_KEYUP:
    repeat = 1;              /* Clears the repeat count  */
    break;
```

7.4.2 Using the Cursor when No Mouse Is Available

When no mouse is available, the application must display and move the cursor in response to keyboard actions. To determine whether a mouse is present, you can use the **GetSystemMetrics** function and specify the SM_MOUSEPRESENT option:

```
GetSystemMetrics(SM_MOUSEPRESENT);
```

This function returns TRUE if the mouse is present.

You will need to display the cursor and update the cursor position when the application is activated, and hide the cursor when the application is deactivated. The following statements carry out both activation functions:

```
case WM_ACTIVATE:
    if (!GetSystemMetrics(SM_MOUSEPRESENT)) {
        if (!HIWORD(lParam)) {
            if (wParam) {
                SetCursor(hMyCursor);
                ptCursor.x = CursorX;
                ptCursor.y = CursorY;
                ClientToScreen(hWnd, &ptCursor);
                SetCursorPos(ptCursor.x, ptCursor.y);
            }
            ShowCursor(wParam);
        }
    }
    break;
```

The cursor functions are called only if the system has no mouse; that is, if the **GetSystemMetrics** function returns FALSE. Since Windows positions and updates the cursor automatically if a mouse is present, the cursor functions, if carried out, would disrupt this processing.

The next step is to determine whether or not the window is iconic. The cursor must not be displayed or updated if the window is an icon. In a WM_ACTIVATE message, the high-order word is nonzero if the window is iconic, so the cursor functions are called only if this value is zero.

The final step is to check the wParam parameter to determine whether the window is being activated or deactivated. This parameter is nonzero if the window is being activated. When a window is activated, the **SetCursor** function sets the shape and the **SetCursorPos** function positions it. The **ClientToScreen** function converts the cursor position to screen coordinates as required by the **SetCursorPos** function. Finally, the **ShowCursor** function shows or hides the cursor depending on the value of the wParam parameter.

When the system has no mouse installed, applications must be careful when using the cursor. In general, applications must hide the cursor when the window is closed, destroyed, or relinquishes control. If an application fails to hide the cursor, it prevents subsequent windows from using the cursor. For example, if an application sets the cursor to the hourglass, displays the cursor, then relinquishes control to a dialog box, the cursor remains on the screen (possibly in a new shape), but cannot be used by the dialog box.

7.5 A Sample Application: Cursor

This sample application illustrates how you incorporate cursors and how you use the mouse and keyboard in your applications. The Cursor application shows how to do the following:

- Use a custom cursor as the class cursor.
- Show the hourglass cursor during a lengthy operation.
- Use the mouse to select a portion of the client area.
- Use the keyboard to move the cursor.

To create the Cursor application, copy and rename the source files of the Generic application, then make the following modifications:

1. Add a **CURSOR** statement to your resource script file.
2. Add new variables.
3. Load the custom cursor and use it to set the class cursor in the initialization function.
4. Add a lengthy operation to the window function (for simplicity, use the ENTER key to trigger the operation).
5. Add the WM_LBUTTONDOWN, WM_MOUSEMOVE, and WM_LBUTTONUP cases to the window function to support selection.
6. Add the WM_KEYDOWN case to the window function to support keyboard-controlled cursor movement.

This sample assumes that your system has a mouse, so if your system does not, the application may not operate as described. However, it is a fairly straightforward task to adjust the sample to work with both the mouse and the keyboard or with only the keyboard.

7.5.1 Add the CURSOR Statement

To use a custom cursor, you need to create a cursor file, using Icon Editor, and give the name of the file in a **CURSOR** statement in the resource script file. Add the following statement to your resource script file:

```
bullseye CURSOR bullseye.cur
```

Make sure that the cursor file, *bullseye.cur*, contains a cursor.

7.5.2 Add New Variables

You will need several new variables for this sample application. Place the following statements at the beginning of your C-language source file:

```
char str[255];                  /* general-purpose string buffer */

HCURSOR hSaveCursor;            /* handle to current cursor      */
HCURSOR hHourGlass;             /* handle to hourglass cursor    */

BOOL bTrack = FALSE;            /* TRUE if left button clicked   */
int OrgX = 0, OrgY = 0;         /* original cursor position      */
int PrevX = 0, PrevY = 0;       /* current cursor position       */
int X = 0, Y = 0;               /* last cursor position          */
RECT Rect;                      /* selection rectangle           */

POINT ptCursor;                 /* x and y coordinates of cursor */
int repeat = 1;                 /* repeat count of keystroke     */
```

The hSaveCursor and hHourGlass variables hold the cursor handles to be used for the lengthy operation. The bTrack parameter holds a Boolean flag indicating whether a selection is in progress. The variables OrgX, OrgY, PrevX, and PrevY hold the original and current mouse positions as a selection is being made. OrgX and OrgY, along with the variables X and Y, hold the original and final coordinates of the selection when the selection process is complete. The ptCursor structure holds the current position of the cursor in the client area. This is updated when the user presses a DIRECTION key. The Rect structure holds the current dimensions of the client area and is used to make sure the cursor stays within the client area. The repeat variable holds the current repeat count for keyboard motion.

7.5.3 Set the Class Cursor

To set the class cursor, you need to modify a statement in the initialization function. Specifically, you need to assign the cursor handle the **hCursor** field of the window-class structure. Make the following change in the C-language source file. Find this line:

```
pWndClass->hCursor = LoadCursor(NULL, IDC_ARROW);
```

Change it to the following:

```
pWndClass->hCursor = LoadCursor(hInstance, "bullseye");
```

7.5.4 Prepare the Hourglass Cursor

Since you will be using the hourglass cursor during a lengthy operation, you need to load it. The most convenient place is to load it in the WM_CREATE case as the window is being created. Add the following statement:

```
hHourGlass = LoadCursor(NULL, IDC_WAIT);
```

This makes the hourglass cursor available whenever it is needed.

7.5.5 Add a Lengthy Operation

A lengthy operation can take many forms. In this sample, it will be a function named "sieve" that computes several hundred prime numbers. The operation begins when the user presses the ENTER key. Add the following statements to the window function:

```
case WM_CHAR:       ') {
    if (wParam == '
    SetCapture(hWnd);

    hSaveCursor = SetCursor(hHourGlass);

    hDC = GetDC(hWnd);
    TextOut(hDC, 1, 1, "Calculating prime numbers...", 28);
    sprintf(str, "Calculated %d primes. ", sieve());
    TextOut(hDC, 1, 1, str, strlen(str));
    ReleaseDC(hWnd, hDC);

    SetCursor(hSaveCursor); /* Restores previous cursor */
    ReleaseCapture();
}
break;
```

When the ENTER key is pressed, Windows generates a WM_CHAR message whose wParam parameter contains an ANSI value representing the carriage-return value. When the window function receives a WM_CHAR message, it checks for this value and carries out the sample lengthy operation, sieve. This function, called Eratosthenes Sieve Prime-Number Program, is from *Byte*, January 1983. It is defined as follows:

```
#define NITER 20
#define SIZE 8190

char flags[SIZE+1] = { 0};
```

```
sieve() {
    int i,k;
    int iter, count;

    for (iter = 1; iter <= NITER; iter++) {
         count = 0;
        for (i = 0; i <= SIZE; i++)
             flags[i] = TRUE;

        for (i = 2; i <= SIZE; i++) {
            if (flags[i] ) {
                for (k = i + i; k <= SIZE; k += i)
                    flags[k] = FALSE;
                count++;
             }
         }
    }
    return (count);
}
```

7.5.6 Add the WM_LBUTTONDOWN, WM_MOUSEMOVE, and WM_LBUTTONUP Cases

To carry out a selection, you can add the statements as described in Section 7.3, "Using the Mouse." Add the following statements to your window function:

```
case WM_LBUTTONDOWN:
    bTrack = TRUE;

    if (OrgX != X || OrgY != Y) {      /* Clears previous box */
       hDC = GetDC(hWnd);
       SetROP2(hDC, R2_NOT);
       MoveTo(hDC, OrgX, OrgY);
       LineTo(hDC, OrgX, Y);
       LineTo(hDC, X, Y);
       LineTo(hDC, X, OrgY);
       LineTo(hDC, OrgX, OrgY);
       ReleaseDC(hWnd, hDC);
    }
    PrevX = LOWORD(lParam);
    PrevY = HIWORD(lParam);
    if (!(wParam & MK_SHIFT)) { /* If shift key is not pressed */
       OrgX = LOWORD(lParam);
       OrgY = HIWORD(lParam);
    }
    else {     /* Shift key is pressed, update the current box */
        hDC = GetDC(hWnd);
        SetROP2(hDC, R2_NOT);
        MoveTo(hDC, OrgX, OrgY);
        LineTo(hDC, OrgX, PrevY);
        LineTo(hDC, PrevX, PrevY);
        LineTo(hDC, PrevX, OrgY);
        LineTo(hDC, OrgX, OrgY);
        ReleaseDC(hWnd, hDC);
     }
```

```
        SetCapture(hWnd);
        break;

case WM_MOUSEMOVE:
    if (bTrack) {
        hDC = GetDC(hWnd);
        SetROP2(hDC, R2_NOT);          /* Erases the previous box */
        MoveTo(hDC, OrgX, OrgY);
        LineTo(hDC, OrgX, PrevY);
        LineTo(hDC, PrevX, PrevY);
        LineTo(hDC, PrevX, OrgY);
        LineTo(hDC, OrgX, OrgY);

        PrevX = LOWORD(lParam);
        PrevY = HIWORD(lParam);
        MoveTo(hDC, OrgX, OrgY);           /* Draws the new box */
        LineTo(hDC, OrgX, PrevY);
        LineTo(hDC, PrevX, PrevY);
        LineTo(hDC, PrevX, OrgY);
        LineTo(hDC, OrgX, OrgY);
        ReleaseDC(hWnd, hDC);
    }
    break;

case WM_LBUTTONUP:
    bTrack = FALSE;             /* Ignores mouse input          */
    ReleaseCapture();          /* Releases hold on mouse input */

    X = LOWORD(lParam);        /* Saves the current value      */
    Y = HIWORD(lParam);
    break;
```

7.5.7 Add the WM_KEYDOWN Case

In order to use the keyboard to control the cursor, you need to add a WM_KEYDOWN case to the window function. The statements in this case should retrieve the current position of the cursor and update the position when a DIRECTION key is pressed. Add the following statements to the window function:

```
case WM_KEYDOWN:
    GetCursorPos(&ptCursor);
    if (wParam != VK_LEFT || wParam != VK_RIGHT ||
        wParam != VK_UP || wParam != VK_DOWN )
        break;

    ScreenToClient(hWnd, &ptCursor);
    repeat++;                   /* Increases the repeat rate */

    switch (wParam) {

        case VK_LEFT:
            ptCursor.x -= repeat;
            break;

        case VK_RIGHT:
            ptCursor.x += repeat;
            break;
```

```
        case VK_UP:
            ptCursor.y -= repeat;
            break;

        case VK_DOWN:
            ptCursor.y += repeat;
            break;

        default:
            return (NULL);
        }

        GetClientRect(hWnd, &Rect); /* Gets the client boundaries */

        if (ptCursor.x >= Rect.right)
           ptCursor.x = Rect.right - 1;
        else if (ptCursor.x < Rect.left)
           ptCursor.x = Rect.left;
        if (ptCursor.y >= Rect.bottom)
           ptCursor.y = Rect.bottom - 1;
        else if (ptCursor.y < Rect.top)
           ptCursor.y = Rect.top;

    case WM_KEYUP:
        repeat = 1;                    /* Clears the repeat count  */
        break;
```

The **GetCursorPos** function retrieves the cursor position in screen coordinates. To check the position of the cursor within the client area, the coordinates are converted to client coordinates by using the **ScreenToClient** function. The **switch** statement checks for the DIRECTION keys and adds the content of the repeat-count field of the lParam parameter to the current position. The new position is checked to make sure it is still in the client area, using the **GetClientRect** function to retrieve the dimensions of the client area. The position is adjusted, if necessary. Finally, the **ClientToScreen** function converts the position back to screen coordinates and the **SetCursorPos** function sets the new position.

7.5.8 Compile and Link

No changes are required to the **make** file to recompile and link the Cursor application. When the application is recompiled, start Windows and the Cursor application. When you move the cursor into the client area it should look like Figure 7.2:

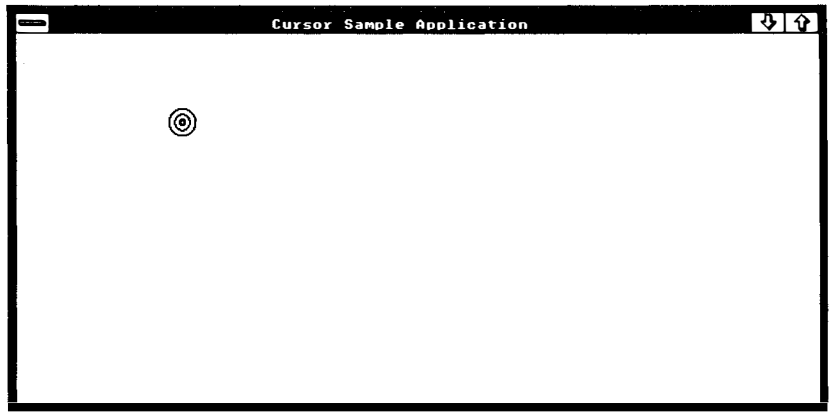


Figure 7.2 Cursor Window

Press and hold down the left mouse button, then drag the mouse to a new position and release the mouse button. You should see a selection that looks like Figure 7.3:

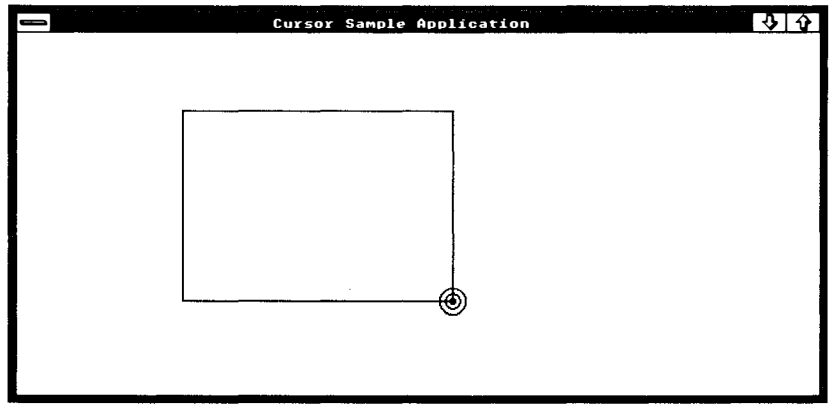


Figure 7.3 A Selection in Cursor

Now press the DIRECTION keys to move the cursor. Then press the ENTER key to view the lengthy operation.

Chapter 8
Menus

8.1 What Are Menus?

A menu is a list of items, called menu items, which are names or bitmaps that represent actions an application can take. Menu items are the application's command names. The user can direct the application to carry out a command by using either the mouse or the keyboard to choose the corresponding menu item. When a user chooses a command, Microsoft Windows sends a message to the application specifying which command was chosen.

This chapter shows how to create menus for your applications and how to process input from menus. It also shows how to use keyboard accelerators and command mnemonics with menus. Finally, it shows how to modify menus, use bitmaps with menus, and make special use of the application's system menu.

8.2 Using Menus

You can use a menu in any overlapped or pop-up window, but not in a child window. To use a menu, you can register a class menu, which is applied by default whenever you create a window of that class, or you can explicitly specify a menu when you create the window. Before using a menu, however, you must define it in your application's resource script file.

8.2.1 Defining a Menu

You can define the content of a menu by using a **MENU** statement in the application's resource script file. The following **MENU** statement illustrates a simple class menu:

```
# define IDM_COMMAND1    1
# define IDM_COMMAND2    2
# define IDM_COMMAND3    3
# define IDM_COMMAND4    4

SimpleMenu MENU
BEGIN
    MENUITEM "Command1", IDM_COMMAND1
    MENUITEM "Command2", IDM_COMMAND2
    POPUP "Menu1"
    BEGIN
        MENUITEM "Command3", IDM_COMMAND3
        MENUITEM "Command4", IDM_COMMAND4
    END
END
```

The menu, named "SimpleMenu", contains two commands: Command1 and Command2, and a single pop-up menu, Menu1. The Menu1 menu contains two commands: Command3 and Command4. Each command has a unique identifier, or menu ID. Windows passes the menu ID of a command to the application when the user chooses the command. Menu IDs must be unique constants.

8.2.2 Setting the Class Menu

The default menu for any window is the class menu. You define the class menu when you register the window class. To define a class menu, you assign the name of the menu, as given in the resource file, to the **lpszMenuName** field of the window-class structure. In the following statement, the class menu is named "SimpleMenu":

```
pWndClass->lpszMenuName = (LPSTR) "SimpleMenu"
```

In this example, the pWndClass variable is assumed to point to a **WndClass** data structure. The menu name is the name given to the menu in the application's resource file.

Once a class menu has been registered, each window of that class will have a class menu unless you override the default by explicitly supplying a menu handle when you create the window.

8.2.3 Setting a Window Menu

You don't have to use the class menu for a window. Instead, you can set a window's menu when you create the window by specifying a menu handle with the **CreateWindow** function. A menu handle, returned by the **CreateMenu** or **LoadMenu** function, identifies the menu. The following example shows how to load and specify a menu by using **CreateWindow**:

```
HWND hWnd;
HMENU hMenu;
    .
    .
    .

hMenu  = LoadMenu(hInstance, "SimpleMenu");
hWnd = CreateWindow("Sample",
           "Sample",
           WS_OVERLAPPEDWINDOW,
           CW_USEDEFAULT,
           CW_USEDEFAULT,
           CW_USEDEFAULT,
           CW_USEDEFAULT,
           (HWND) NULL,
           hMenu,
           hInstance,
           (LPSTR) NULL );
```

The **LoadMenu** function loads the menu named "SimpleMenu". The hInstance variable specifies that the resource is to be loaded from the application's resources. The returned menu handle is used in the **CreateWindow** function to override the class menu, if one exists.

8.3 Modifying Menus

You can modify a menu at any time, or even create new menus, by using the **EnableMenuItem**, **CheckMenuItem**, **CreateMenu**, and **ChangeMenu** functions. These functions let you enable and disable a menu item; put a checkmark by a menu item; add, delete, or modify a menu item; and even replace an entire menu. These functions also let you modify a menu, as necessary, during the execution of an application.

Modifying a menu for a window that uses the class menu will not affect the menus of other windows in the same class. When a window is created, it receives a private copy of the class menu. It can change this private copy without affecting the menus of other windows.

8.3.1 Replacing a Menu

You can replace a window's menu by using the **SetMenu** function. The **SetMenu** function is typically used when the application changes modes and needs a completely new set of commands. For example, an application could replace a spreadsheet menu with a charting menu when the user changes from a spreadsheet to a charting mode.

In the following example, the **GetMenu** function retrieves the current menu handle from the specified window and saves it for restoring later. The **SetMenu** function replaces the current menu with a customized menu loaded from the application's resources.

```
HMENU hMenu;
HMENU hOldMenu;
     .
     .
     .
hOldMenu = GetMenu(hWnd);
hMenu = LoadMenu(hInstance, "CustomMenu");
SetMenu(hWnd, hMenu);
     .
     .
     .
```

The customized menu can also be loaded from resources other than those belonging to the application (by using the module handle of a library), or could be created in memory by using the **CreateMenu** and **ChangeMenu** functions.

8.3.2 Enabling/Disabling Menu Items

You can enable or disable a menu item by using the **EnableMenuItem** function. This function enables, disables, or grays a command. A disabled command looks unchanged, but does not respond to mouse clicks or selection by the keyboard. A grayed command has a grayed command name and does not respond to mouse clicks or selection by the keyboard. You typically disable or gray a menu item when the action it represents is not appropriate. For example, you can gray the Print command in the File menu when there is no printer currently installed in the system.

The following example disables the command whose menu ID is IDM_SAVE:

```
EnableMenuItem(hMenu, IDM_SAVE, MF_DISABLED);
```

The following example disables the command whose menu ID is IDM_PRINT and redisplays the command name in gray letters:

```
EnableMenuItem(hMenu, IDM_SAVE, MF_GRAYED);
```

You can enable a command or menu by using the MF_ENABLED option. If you change menus or commands in the menu bar, you will need to call the **DrawMenuBar** function to display the changes. In the following example, the command identified by ID_EXIT is enabled and the menu bar is redrawn to display the change:

```
EnableMenuItem(hMenu, ID_EXIT, MF_ENABLED);
DrawMenuBar(hMenu);
```

You can specify the initial state of a menu or command by using the **INACTIVE** and **GRAYED** options with the **MENUITEM** statement in your resource script file. For example, the following statement sets the initial state of the Print command to grayed:

```
MENUITEM "Print", IDM_PRINT, GRAYED
```

The option applies only to the initial state of the menu. You can change the state by using the **EnableMenuItem** function in your C-language source file.

8.3.3 Checking Menu Items

You can place or remove a checkmark next to a specified menu item by using the **CheckMenuItem** function. You typically check a menu item when it is in a group of commands that are mutually exclusive. The checkmark indicates the user's latest choice. For example, if the group of commands is Left, Right, and Center, you can put a checkmark by Left to show that it is the latest command chosen by the user.

The following example places a checkmark next to the item whose menu ID is IDM_LEFT:

```
CheckMenuItem(hMenu, IDM_LEFT, MF_CHECKED);
```

The following example removes the check (if any) from the item whose menu ID is IDM_RIGHT:

```
CheckMenuItem(hMenu, IDM_RIGHT, MF_UNCHECKED);
```

If you change menu items in the menu bar, you will need to call the **DrawMenuBar** function to display the changes.

You can place an initial checkmark next to a command by using the **CHECKED** option in the **MENUITEM** statement in the resource file. The following example shows how to make an initial checkmark in the resource file:

```
MENUITEM "Left", IDM_LEFT, CHECKED
```

This option applies only to the initial state of the menu. You can change the state by using the **CheckMenuItem** function in your C-language source file.

8.3.4 Using the WM_INITMENU Message

Windows sends a WM_INITMENU message to the window function owning a menu just before Windows displays the menu. This lets the window function check and modify the state of the menu items before the menu is displayed. In the following example, the window function processes the WM_INITMENU message, setting the state of a command based on the value of the wChecked variable:

```
WORD wChecked = IDM_LEFT;
    .
    .
    .
case WM_INITMENU:
    if (GetMenu(hWnd) != wParam)
        break;
    CheckMenuItem(wParam, IDM_LEFT,
        IDM_LEFT == wChecked ? MF_CHECKED : MF_UNCHECKED);
    CheckMenuItem(wParam, IDM_CENTER,
        IDM_CENTER == wChecked ? MF_CHECKED : MF_UNCHECKED);
    CheckMenuItem(wParam, IDM_RIGHT,
        IDM_RIGHT == wChecked ? MF_CHECKED : MF_UNCHECKED);
    break;
```

The WM_ INITMENU message passes the given menu handle in the wParam parameter. To make sure the menu about to be displayed is the window's menu, the **GetMenu** function retrieves a handle to the current window's menu that can be compared with wParam. If these are not equal, the window's menu should not be initialized. Otherwise, you can use the **CheckMenuItem** function to initialize the commands in the menu.

The following expression, and the others like it in this example, are quick ways to check the value of the latest command, and to choose the correct initializing action based on that value:

```
(IDM_LEFT == wChecked) ? MF_CHECKED : MF_UNCHECKED;
```

8.3.5 Changing Existing Menus

You can change the appearance and order of menus and menu items by using the **ChangeMenu** function. The function can add items to new or existing menus and change the status of new or existing menu items.

In the following example, the **ChangeMenu** function adds a pop-up menu named "Go" to the window's menu (the Go menu lists four commands: Home, Work, Store, and Park):

```
HMENU hMenu;
HMENU hPopupMenu;
    .
    .
    .

hMenu = GetMenu(hWnd);
hPopupMenu = CreateMenu();

ChangeMenu(hMenu,
    2,
    "&Go",
    hPopupMenu,
    MF_CHANGE | MF_POPUP | MF_BYPOSITION);

ChangeMenu(hPopupMenu,
    0,
    "&Home",
    IDM_CAR,
    MF_APPEND | MF_STRING | MF_BYCOMMAND);

ChangeMenu(hPopupMenu,
    0,
    "&Work",
    IDM_HOUSE,
    MF_APPEND | MF_STRING | MF_BYCOMMAND);

ChangeMenu(hPopupMenu,
    0,
    "&Store",
    IDM_CLOTHES,
    MF_APPEND | MF_STRING | MF_BYCOMMAND);
```

```
ChangeMenu(hPopupMenu,
    0,
    "&Park",
    IDM_SERVICES,
    MF_APPEND | MF_STRING | MF_BYCOMMAND);
```

In this example, the **GetMenu** function retrieves a handle to the menu of the window identified by the hWnd variable. The **CreateMenu** function creates a new menu whose handle is assigned the hPopupMenu variable. The first **ChangeMenu** function adds the new menu to the window menu. The MF_CHANGE, MF_POPUP, and MF_BYPOSITION flags specify that **ChangeMenu** should replace (MF_CHANGE) the existing menu at position 2—the third item from the left end of the menu (MF_BYPOSITION)—with the new pop-up menu (MF_POPUP).

The next four **ChangeMenu** functions append four menu items to the new pop-up menu. In each case, the pop-up menu is identified by its handle, hPopupMenu. Since the item is being appended, NULL is given as the item's current position. The name of the menu item, with its mnemonic, appears next. The menu ID of the new item follows. Finally, the MF_APPEND, MF_STRING, and MF_BYCOMMAND options specify the operation to carry out: append the menu-item name (a string) to the given menu and set the new menu ID by command value.

8.3.6 Adding Bitmaps to Menus

You can also use bitmaps as menu items. This can be done with the **ChangeMenu** function by simply using the MF_BITMAP option and specifying the bitmap to be used. To add a bitmap to a menu, you need to create a bitmap or load one from the application's resources. You can then replace an existing menu item or append the bitmap to the end of the menu.

In the following example, a bitmap named "dog" is loaded and used in the **ChangeMenu** function to add it to the window's menu:

```
HMENU hMenu;
HANDLE hBitmap;
    .
    .
    .
hBitmap = LoadBitmap(hInstance, "dog");

hMenu = GetMenu(hWnd);
ChangeMenu(hMenu,
    IDM_PARK,
    MAKELONG(hBitmap, 0),
    IDM_PARK,
    MF_CHANGE | MF_BYCOMMAND | MF_BITMAP);
```

The **LoadBitmap** function loads the bitmap from the file and returns a handle to the bitmap, saved in the hBitmap variable. The **ChangeMenu** function then replaces the name of the existing menu item (identified by IDM_PARK) with the bitmap. The bitmap handle must be passed as the low-order word of the third argument to **ChangeMenu**. The **MAKELONG** utility combines the 16-bit handle with a 16-bit constant to make the 32-bit argument. The MF_BITMAP constant specifies that the low-order word identifies a bitmap.

8.4 Using the System Menu

The system menu is a menu that is usually supplied and maintained by Windows for each window. However, some applications may wish to append additional commands to a window's system menu or even process some of the system-menu commands. This section explains how to use the system menu.

8.4.1 The Default System Menu

For each window, Windows creates a default system menu that contains the commands that can be used with that given type of window. For example, in a pop-up window, the Icon command is grayed since pop-up windows cannot be made iconic. The **GetSystemMenu** function returns a handle to a window's default system menu. An application can, if necessary, change or disable any item in the default system menu. This is typically done when the application initializes its window or receives a WM_INITMENU message.

8.4.2 Changing the System Menu

To change the system menu, you must retrieve a handle to it by using the **GetSystemMenu** function, then use the **ChangeMenu** function to make changes. You can identify system-menu items by command if you use values such as SC_MAXIMIZE and SC_MINIMIZE.

If you have added to or changed the system menu, you must process any WM_SYSCOMMAND messages that correspond to the menu items you added or changed. Windows sends a WM_SYSCOMMAND message to the application when the user selects commands from the system menu. WM_SYSCOMMAND messages for unchanged commands must be sent to the **DefWindowProc** function.

You can restore the system menu to its initial state by calling the **GetSystemMenu** function with the second parameter set to TRUE.

You can check or gray a system-menu item by processing the WM_INITMEN message. Windows sends this message to the application whenever the user selects the system menu (or any other pop-up menu), but before Windows displays the menu. Under some circumstances, Windows automatically grays one or more system-menu commands.

8.4.3 Adding Items to the System Menu

You have already seen how to add an "About..." command to the system menu of the sample application, Generic. In general, you can use the same method to add other menu items to the system menu. If you do, it is recommended that you place a separator between the new item and the existing items. Also, you must not add pop-up menus to the system menu.

In the following example, the **ChangeMenu** function is used to add a separator and the About... menu item to the system menu:

```
#define IDABOUT 100

HMENU hMenu;
    .
    .
    .
HMENU hMenu = GetSystemMenu(hWnd, FALSE);

ChangeMenu(hMenu, 0, NULL, NULL, MF_APPEND | MF_SEPARATOR);
ChangeMenu(hMenu, 0, "A&bout ...", IDABOUT, MF_APPEND | MF_STRING);
```

When adding a menu item to the system menu, you must make sure that the menu ID of that item is greater than any of the system-command (SC_) values defined in the *windows.h* file.

Whenever you add an item to the system menu, you must process the input from that item. When the user chooses an item in the system menu, Windows passes a WM_SYSCOMMAND message, containing the menu ID of the item, to the window function. The window function should pass the standard system-menu message on to **DefWindowProc** and carry out its own processing for the added items. In the following example, the window function picks out the WM_SYSCOMMAND messages for the About command and processes them separately:

```
switch (message) {
    .
    .
    .
    case WM_SYSCOMMAND:
        switch (wParam) {
            case IDABOUT:
                .
                .
                .
                break;
```

```
        default:
            return (DefWindowProc(hWnd, message, wParam, lParam);
    }
    break;
```

8.4.4 System-Menu Accelerators

Applications can associate keyboard accelerators with system-menu commands. When the user presses such an accelerator, Windows sends a WM_SYSCOMMAND message to the application.

You can associate accelerators with system-menu commands by creating an accelerator table in your resource script file that uses the menu ID (whose low-order byte must be greater than any SC_ constant) with the accelerator keys. The following example illustrates how to create an accelerator table that associates accelerators with the system-menu commands:

```
acceltab ACCELERATORS
BEGIN
        ^B, IDM_ABOUT
END
```

Accelerators for system-menu commands generate WM_SYSCOMMAND messages when the window is iconic—as long as the icon is active.

8.5 A Sample Application: FileMenu

Applications that open and save data files use the File menu to offer commands that direct the application to open, save, and clear files. This sample application, FileMenu, shows how to add a File menu to an application and how to process input from a menu (FileMenu does not show how to open and save files, or how to prompt for user input. These tasks are described in later chapters in this guide.). To create FileMenu, you will need to copy and rename the Generic source files. Then do the following:

1. Add a File menu to the resource file.
2. Add definitions to the include file.
3. Add a WM_COMMAND case to the window function.
4. Remove the WM_CREATE and WM_SYSCOMMAND cases supporting the About command in the system menu.
5. Compile and link the application.

Since the File menu is typically the first menu in an application, the About command, previously located in the system menu, should now be appended to the end of the File menu. Also, you need to add an Exit command to the File menu. It should appear immediately before the About command.

Note

The File menu's purpose and content are defined in the *Microsoft Windows Application Style Guide*. See the style guide for additional information about the File menu.

8.5.1 Add a File Menu to the Resource File

You need to add a **MENU** statement to your resource script file. This statement defines the menu names and commands for your menu. For this sample, add the following **MENU** statement to your resource file:

```
# include "filemenu.h"

#include "windows.h"
#include "filemenu.h"

FileMenu MENU
BEGIN
    POPUP        "&File"
    BEGIN
        MENUITEM    "&New",                 IDM_NEW
        MENUITEM    "&Open...",             IDM_OPEN
        MENUITEM    "&Save",                IDM_SAVE
        MENUITEM    "Save &As...",          IDM_SAVEAS
        MENUITEM    "&Print",               IDM_PRINT
        MENUITEM    SEPARATOR
        MENUITEM    "E&xit",                IDM_EXIT
        MENUITEM    "A&bout Filemenu...", IDM_ABOUT
    END
END
```

The **MENU** statement creates the File menu. The File menu is a pop-up menu (defined by the **POPUP** statement) that contains five commands: New, Open, Save, Save As, and Print. Commands are defined by the **MENUITEM** statement. The Open and Save As command names include ellipses (...) to indicate that the commands require additional information and will prompt for that information. Each command has a unique menu identifier: IDM_NEW, IDM_OPEN, IDM_SAVE, IDM_SAVEAS, and IDM_PRINT. The menu ID is used by the application to identify the menu when the user chooses it.

Notice that each menu and command name includes an ampersand (&). The ampersand specifies the menu or command mnemonic. For more information about mnemonics, see Chapter 3, "A Sample Application: Generic."

8.5.2 Add Definitions to the Include File

The menu IDs must be declared in your include file since these constants are used in both the C-language source and in the resource script file. For the File menu, add the following statements to your include file:

```
#define     IDM_NEW       100
#define     IDM_OPEN      101
#define     IDM_SAVE      102
#define     IDM_SAVEAS    103
#define     IDM_PRINT     104
#define     IDM_EXIT      105
#define     IDM_ABOUT     106
```

A menu ID can have any integer value. The only restriction is that menu IDs must be unique within a menu. That is, no two commands in a menu can have the same menu ID.

8.5.3 Add the WM_COMMAND Case

The easiest way to incorporate a menu into an application is to register the menu as a class menu. Once the menu is registered, Windows automatically loads it from the resource file when it is needed; for example, when a window using the class menu is created. For this sample, you need to add the following statement in the **FileMenuInit** function, before the call to the **RegisterClass** function:

```
pWndClass->lpszMenuName = (LPSTR)"FileMenu";
```

This statement copies a pointer to the menu into the window-class structure. Windows will use this pointer to specify the name of the class menu when it loads the menu from the resource file.

When a user chooses a command in a menu, Windows sends a WM_COMMAND message to the corresponding window function. The message contains the menu ID of the command chosen. The menu ID for the command is sent in the wParam parameter. It is up to the window function to carry out any tasks associated with the command. For example, if the user chooses the Open command, the window function is responsible for prompting for the filename, opening the file, and displaying the file in the window's client area.

This sample shows only how to begin to process menu input. Instead of performing tasks, the New, Open, Save, Save As, and Print commands activate a "Command not implemented" message box. The Exit command calls the **DestroyWindow** function to destroy the window and terminate the application. The About FileMenu command displays the About dialog box. This is the same action that was previously carried out by the WM_SYSCOMMAND case. The WM_SYSCOMMAND case is no longer needed and should be discarded. You need only add the following case statement to the window function:

```
case WM_COMMAND:
    switch (wParam) {

        case IDM_NEW:
        case IDM_OPEN:
        case IDM_SAVE:
        case IDM_SAVEAS:
        case IDM_PRINT:
            MessageBox(hWnd, "Command not implemented",
                       (LPSTR) NULL, MB_OK);
            break;

        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;

        case IDM_ABOUT:
            lpProcAbout = MakeProcInstance(About, hInst);
            DialogBox(hInst, "AboutBox", hWnd, lpProcAbout);
            FreeProcInstance(lpProcAbout);
            break;
    }
    break;
```

8.5.4 Delete the WM_CREATE and WM_SYSCOMMAND Cases

Since the WM_COMMAND case supports the About FileMenu... command, the WM_CREATE and WM_SYSCOMMAND cases are no longer needed and should be deleted. In particular, you should delete the statements in the WM_CREATE case that change the system menu. The entire WM_SYSCOMMAND case should be deleted since special processing of the system menu input is not required.

8.5.5 Compile and Link

No changes are required to the **make** file to recompile and link the FileMenu application. Start Windows, then start the FileMenu application and select the File menu; for example, click the menu name, File, with the mouse. The menu will look like Figure 8.1:

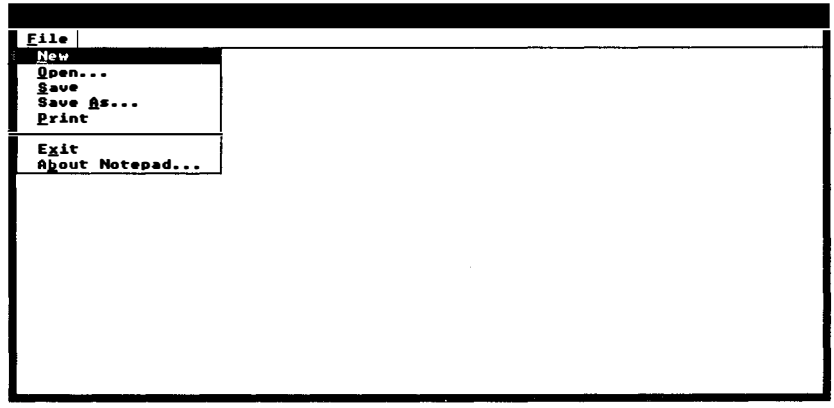


Figure 8.1 FileMenu Window

If you choose any of the commands in the File menu, you will get the message shown in Figure 8.2:

Figure 8.2 FileMenu Error Message

8.6 A Sample Application: EditMenu

The EditMenu sample application illustrates the second most common menu, the Edit menu. The Edit menu lets the user direct an application to carry out clipboard operations, such as copying and pasting within the application or between two cooperating applications. Creating and processing an Edit menu is similar to creating and processing the File menu, except that the Edit menu also includes accelerator keys. These are shortcut keys that let the user choose a command from a menu by using a single keystroke. Many commands, such as Edit commands, are more convenient when initiated from the keyboard.

Accelerator keys are provided as part of the resource file, and are tied into the application through the C-language source code. The keys shown in this sample are specifically reserved and should be used only as accelerator keys for the Edit menu.

To create the EditMenu application, copy and rename the FileMenu source files. Then do the following:

1. Add an Edit menu to the resource file.
2. Add an accelerator table to the resource file.
3. Add a new variable.
4. Load the accelerator table.
5. Modify the message loop in **WinMain**.
6. Modify the WM_COMMAND case.
7. Compile and link the application.

EditMenu does not show how to use the clipboard. This task is described in Chapter 13, "The Clipboard."

Note

The Edit menu's purpose and content are defined in the *Microsoft Windows Application Style Guide*. See the style guide for additional information about the Edit menu.

8.6.1 How Accelerator Keys Work

To use accelerators, an application must first create an accelerator table that contains a list of the accelerator keys, then use the accelerator table to translate keyboard messages into menu input. An accelerator table is a list of keystrokes and corresponding menu IDs that you create by using the **ACCELERATORS** statement in the resource file. To use the accelerator table in the application, you must load it by using the **LoadAccelerators** function, then you must pass each message you receive through the table by using the **TranslateAccelerator** function, which compares the message with the keystrokes listed in the table. If there is a match, the function creates a WM_COMMAND message and sends it to the window function that is using the corresponding menu ID. The window function can then carry out the command as if the user had chosen it from the menu.

8.6.2 Add an Edit Menu to the Resource File

You need to add an Edit menu to the **MENU** statement in the resource file. Since the File menu should be the first menu on the menu bar, you should place the Edit-menu definition immediately after the File-menu definition. The **MENU** statement should now look like this:

```
EditMenu MENU
    POPUP "&File"
    BEGIN
    .
    .
    .
    END

    POPUP        "&Edit"
    BEGIN
        MENUITEM    "&UndoALT+BKSP",      IDM_UNDO
        MENUITEM    SEPARATOR
        MENUITEM    "Cu&tShift+Del",      IDM_CUT
        MENUITEM    "&CopyCtrl+Ins",      IDM_COPY
        MENUITEM    "&PasteShift+Ins",    IDM_PASTE
        MENUITEM    "C&learDel",          IDM_CLEAR
    END
END
```

The Edit menu has five commands and a separator. The five commands are, Undo, Cut, Copy, Paste, and Clear. Each command has a mnemonic, indicated by the ampersand (&), and an accelerator key, separated from the name with a tab ("\t"). Whenever a command has a corresponding accelerator, it should be displayed in this way. The five accelerator keys in this sample are, ALT+BKSP, SHIFT+DEL, CTRL+INS, SHIFT+INS, and DEL. The separator between the Undo and Cut commands places a horizontal bar between these commands in the menu. A separator is recommended between menu commands that otherwise have nothing in common. For example, Undo affects only the application, whereas the remaining commands affect the clipboard.

8.6.3 Add an Accelerator Table to the Resource File

You can add an accelerator table to the resource file by using the **ACCELERATORS** statement. The statement contains a list of the accelerator keys and the menu IDs of the commands that are associated with the accelerators. Add the following statement to the resource file:

```
EditMenu ACCELERATORS
BEGIN
    VK_BACK,   IDM_UNDO,  VIRTKEY, ALT
    VK_DELETE, IDM_CUT,   VIRTKEY, SHIFT
    VK_INSERT, IDM_COPY,  VIRTKEY, CONTROL
    VK_INSERT, IDM_PASTE, VIRTKEY, SHIFT
    VK_DELETE, IDM_CLEAR, VIRTKEY
END
```

The **ACCELERATORS** statement defines the accelerator keys. As with other resource statements, **BEGIN** starts the entry and **END** marks its end. Five accelerator keys are defined, one for each command. Four accelerators are keystroke combinations using the ALT, SHIFT, or CONTROL key in combination with another key. The keystrokes are defined using the Windows virtual-key code, as indicated by the **VIRTKEY** option. Virtual keys are device-independent key values that Windows translates for each computer. It is a way to guarantee that the same key is used on all computers without knowing what the actual value of the key is on any computer. You may also use ASCII keycodes for accelerators, in which case, you would use the **ASCII** option.

The **ACCELERATORS** statement associates each accelerator with a menu ID. The IDM_UNDO, IDM_CUT, IDM_COPY, IDM_PASTE, and IDM_CLEAR constants are the menu IDs of the Edit-menu commands. When the user presses an accelerator key, these are the values that are passed to the window function.

8.6.4 Add a New Variable

Add the following statement to the beginning of the source file:

```
HANDLE hAccTable;       /* handle to accelerator table */
```

The hAccTable variable is a handle to the accelerator table. It receives the return value of the **LoadAccelerators** function and is used in the **TranslateAccelerator** function to identify the accelerator table.

8.6.5 Load the Accelerator Table

The accelerator table, like any other resource, needs to be loaded before it can be used. You can load the accelerator table in the WM_CREATE case by adding the following statements:

```
case WM_CREATE:
    hAccTable = LoadAccelerators(hInst, "EditMenu");
    break;
```

This statement loads the accelerator table into memory and assigns the handle identifying the table to the hAccTable variable. The hInstance variable identifies the application's resource file, and "EditMenu" is the name of the accelerator table. Once the table is loaded, it can be used in the **TranslateAccelerator** function.

8.6.6 Modify the Message Loop

To use the accelerator table, you must first add a **TranslateAccelerator** function to the message loop. After the function has been added, the message loop should look like this:

```
while (GetMessage(&msg, NULL, NULL, NULL)) {

    if (!TranslateAccelerator(hWnd, hAccTable, &msg)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

The new message loop checks each message to see whether it is an accelerator-key message. If it is, the **TranslateAccelerator** function converts the keystroke to a WM_COMMAND message, which is sent to the window function. If the message is an accelerator, it must not be processed by the **TranslateMessage** and **DispatchMessage** functions. If the message is not an accelerator, it must be processed as usual.

TranslateAccelerator also translates accelerators for commands chosen from the system menu. In such cases, it translates the message into a WM_SYSCOMMAND message. The window handle, hWnd, identifies the window whose messages are to be translated. It must identify the same window whose menu contains the accelerators. The accelerator handle, hAccTable, specifies the accelerator table to be used to translate the accelerator.

8.6.7 Modify the WM_Command Case

You need to process Edit menu commands. In this application, instead of performing tasks, all Edit menu commands activate a "Command not implemented" message box. You need only add the following statements to the WM_COMAMND case:

```
case IDM_UNDO:
case IDM_CUT:
case IDM_COPY:
case IDM_PASTE:
case IDM_CLEAR:
    MessageBox(hWnd, "Command not implemented",
        (LPSTR) NULL, MB_OK);
    break;
```

8.6.8 Compile and Link

No changes are required to the **make** file to compile and link the EditMenu application. Start Windows, then the EditMenu application, and, without opening the pop-up menus, press any of the five accelerator keys. You will notice that the "Command not implemented" message appears when a command is chosen.

Chapter 9
Bitmaps

9.1 What Are Bitmaps?

Bitmaps are bit-pattern images saved in memory for display on an output device. You can use bitmaps in your applications to display images that are otherwise too complex to draw using GDI output functions. This chapter shows how to create and display bitmaps for monochrome as well as color displays. It also shows you how to create and use a background brush.

9.2 Creating Bitmaps

You create a bitmap by supplying GDI with the dimensions and color format of the bitmap, and, optionally, the initial value of the bitmap bits. GDI then returns a handle to the bitmap. You can use this handle in subsequent GDI functions to select and display the bitmap.

You can create bitmaps in three ways:

- Using Icon Editor, create the bitmap, then add it to your application's resources, and load it by using the **LoadBitmap** function.
- Using the **CreateCompatibleBitmap** function, create the bitmap, then select it into a memory device context, and use GDI output functions to draw the bitmap bits.
- Using the **CreateBitmap** function, create the bitmap, then initialize its bits by supplying an array of bits.

The following sections explain how to use each of these methods to create bitmaps.

9.2.1 Creating and Loading Bitmap Files

You can create bitmaps with the Windows 2.0 Icon Editor. Icon Editor lets you specify the dimensions of a monochrome bitmap, then fill it in by setting the individual pixels. A monochrome bitmap consists of an array of bits—one bit representing each pixel on the screen. Since each bit in the bitmap can only be on or off, the bitmap often appears as it would on a monochrome display, just black and white. (On color displays, monochrome bitmaps can be displayed in colors other than black and white. For details, see Section 9.3.2, "Adding Color to Monochrome Bitmaps.")

Figure 9.1 shows a bitmap (*cat.bmp*) being created in Icon Editor:

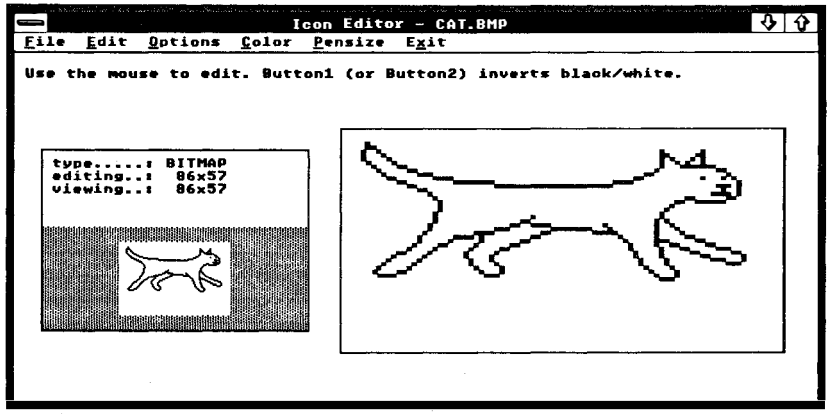


Figure 9.1 A Bitmap in Icon Editor

To create a bitmap, start Icon Editor and follow the directions given in *Microsoft Windows Programming Tools*. After you have created the bitmap, you should save it in a file that has the extension *.bmp*. You may then use the filename in a resource script file to add the bitmap to your application's resources.

To use the bitmap in a bitmap file, you need to add the file to your application's resources, then load it by using the **LoadBitmap** function. The function takes the bitmap's resource name, loads the bitmap into memory, and returns a handle. You can then select the bitmap into a device context and display it by using the **BitBlt** function.

Before loading a bitmap, you must add the bitmap to your application's resources by inserting a **BITMAP** statement to your resource script file. For example, you would use the following statement to load a bitmap named "dog":

```
dog BITMAP dog.bmp
```

The name "dog" identifies the bitmap, and the filename *dog.bmp* specifies the file that contains the bitmap.

To load the bitmap you need to supply the resource name and the instance handle to identify the application's resources:

```
hBitmap = LoadBitmap(hInstance, "dog");
```

The **LoadBitmap** function loads the resource named "dog", then returns a handle to the bitmap. This handle can be used in subsequent GDI functions to select or display the bitmap.

9.2.2 Creating and Drawing Bitmaps

You can create bitmaps by having GDI create a blank bitmap that you can fill in by using GDI output functions. You create a blank bitmap by using the **CreateCompatibleBitmap** function. This function creates a bitmap that has the same format (color planes and bits per pixel) as the given device, but that has no initial image. You can then select the bitmap into a memory device context and use GDI output functions to draw in the bitmap image. This method is often used to create bitmaps that are customized for a particular device.

The following example creates a "star" bitmap by first making a bitmap that is compatible with the display; it then fills the compatible bitmap by using the **Polygon** function:

```
HDC hDC;
HDC hMemoryDC;
HBITMAP hBitmap;
HBITMAP hOldBitmap;
POINT Points[5] = { 32,0, 16,63, 63,16, 0,16, 48,63 };
    .
    .
    .
hDC = GetDC(hWnd);

hMemoryDC = CreateCompatibleDC(hDC);

hBitmap = CreateCompatibleBitmap(hDC, 64, 64);

hOldBitmap = SelectObject(hMemoryDC, hBitmap);

PatBlt(hMemoryDC, 0, 0, 64, 64, WHITENESS);

Polygon(hMemoryDC, Points, 5);

SelectObject(hMemoryDC, hOldBitmap);

DeleteDC(hMemoryDC);

ReleaseDC(hWnd, hDC);
```

The **GetDC** function retrieves a handle to the display context. The bitmap is to be compatible with the display. If you want a bitmap that is compatible with some other device, you should use the **CreateDC** function to retrieve a handle to that device. The **CreateCompatibleDC** function creates the memory device context in which the image of the bitmap will be drawn. The **CreateCompatibleBitmap** function creates the blank bitmap. The size of the bitmap is set to 64 by 64 pixels. The actual number of bits in the bitmap depends on the color format of the display. If the display is a color display, the bitmap will be a color bitmap and may

have many bits for each pixel. The **SelectObject** function selects the bitmap into the memory device context and prepares it for drawing. The handle of the previously selected bitmap is saved.

The **PatBlt** function clears the bitmap and sets all pixels white. This, or a similar function, is required since, initially, the image in a blank bitmap is undefined. You cannot depend on having a clean bitmap in which to draw. The **Polygon** function draws the star by using the endpoints specified in the array of structures, Points.

The subsequent **SelectObject** and **DeleteDC** functions restore the previous bitmap and delete the memory device context. Once the bitmap has been drawn, the memory device context is no longer needed. It is an error to attempt to delete a device context when any bitmap other than the context's original bitmap is selected. Finally, the **ReleaseDC** function releases the display context. The bitmap handle, hBitmap, may now be used in subsequent GDI functions.

9.2.3 Creating Bitmaps with Hard-Coded Bits

You can create a bitmap and set its initial image to an array of bitmap bits by using the **CreateBitmap** function. The function creates a bitmap having an explicit size and color format, then initializes the bitmap image by copying the bits in the specified array to the bitmap. This method is typically used to create small, monochrome bitmaps for use with pattern brushes, but it can also be used to create large bitmaps and color bitmaps.

The following example creates a 64-by-32-pixel, monochrome bitmap and initializes it by using the bits in the array Square:

```
HBITMAP hBitmap;
short Square[] = {
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0xffff,0xffff,0x0000,
0x0000,0xffff,0xffff,0x0000,
0x0000,0xffff,0xffff,0x0000,
0x0000,0xffff,0xffff,0x0000,
0x0000,0xffff,0xffff,0x0000,
0x0000,0xffff,0xffff,0x0000,
0x0000,0xffff,0xffff,0x0000,
0x0000,0xffff,0xffff,0x0000,
0x0000,0xffff,0xffff,0x0000,
0x0000,0xffff,0xffff,0x0000,
```

```
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000 };
    .
    .
    .
hBitmap = CreateBitmap(64, 32, 1, 1, Square);
```

The **CreateBitmap** function creates and initializes the bitmap before returning the bitmap handle. The width and height of the bitmap are 64 and 32 pixels, respectively. The bitmap has one color plane and one bit for each pixel. This means it is a monochrome bitmap.

The Square array contains the bits used to initialize the bitmap. GDI requires that such an array contain short integers; that is, each element should be a 16-bit value. **CreateBitmap** will use as many integers as necessary to fill in each row of the bitmap. If the bitmap is less than 16 bits wide, for example, the function uses the first integer for the first row, discarding any unused bits. When starting a new row, **CreateBitmap** always starts with a new integer.

Once you have created and initialized the bitmap, you can use its handle in subsequent GDI functions. If you want to change the bitmap, you can draw in it by selecting it into a memory device context as described in Section 9.2.2, "Creating and Drawing Bitmaps." If you want to replace the bitmap image with another, you can use the **SetBitmapBits** function to copy another array of bits into the bitmap. For example, the following function call replaces the current bitmap image with the bits in the array Circle:

```
short Circle[] = {
    .
    .
    .
};

SetBitmapBits(hBitmap, 256, Circle);
```

The **SetBitmapBits** function copies the bits in the Circle array into the bitmap specified by the hBitmap variable. The array contains 256 bytes, representing the image of a 64-by-32-pixel monochrome bitmap. If you want to retrieve the current bits in a bitmap before replacing them, you can use the **GetBitmapBits** function. It copies a specified number of bytes from the bitmap into an array of integers.

You can also use the **CreateBitmap** function to create color bitmaps for color displays. However, the format of a color bitmap depends on the corresponding device. This means that if you wish to initialize the bits of the bitmap by using an array, you will need to know the explicit color format used by the device.

9.2.4 Drawing Color Bitmaps

Since hard-coding a color bitmap is device-dependent and may require considerable effort, a better way to create a color bitmap is to create a compatible bitmap and draw in it. For example, to create a color bitmap that has a red, green, and blue plaid pattern, you simply create a blank bitmap and use the **PatBlt** function, with the red, green, and blue brushes, to draw the pattern. This method has the advantage of generating a reasonable bitmap even if the display is a monochrome display that does not support color. This is a result of GDI providing "dithered" brushes for monochrome displays when a color brush is requested. A dithered brush has a unique pattern of pixels that represents a given color when that color is not available for the given device.

The following statements create the color bitmap by drawing it:

```
# define PATORDEST 0x00FA0089L
HDC hDC;
HDC hMemoryDC;
HBITMAP hBitmap;
HBITMAP hOldBitmap;
HBRUSH hRedBrush;
HBRUSH hGreenBrush;
HBRUSH hBlueBrush;
HBRUSH hOldBrush;
     .
     .
     .
hDC = GetDC(hWnd);
hMemoryDC = CreateCompatibleDC(hDC);
hBitmap = CreateCompatibleBitmap(hDC, 64, 32);
hOldBitmap = SelectObject(hMemoryDC, hBitmap);

hRedBrush = CreateSolidBrush(RGB(255,0,0));
hGreenBrush = CreateSolidBrush(RGB(0,255,0));
hBlueBrush = CreateSolidBrush(RGB(0,0,255));

PatBlt(hMemoryDC, 0, 0, 64, 32, BLACKNESS);
hOldBrush = SelectObject(hMemoryDC, hRedBrush);
PatBlt(hMemoryDC, 0, 0, 24, 11, PATORDEST);
PatBlt(hMemoryDC, 40, 10, 24, 12, PATORDEST);
PatBlt(hMemoryDC, 24, 22, 24, 11, PATORDEST);
hOldBrush = SelectObject(hMemoryDC, hGreenBrush);
PatBlt(hMemoryDC, 24, 0, 24, 11, PATORDEST);
PatBlt(hMemoryDC, 0, 10, 24, 12, PATORDEST);
PatBlt(hMemoryDC, 40, 22, 24, 11, PATORDEST);
hOldBrush = SelectObject(hMemoryDC, hBlueBrush);
PatBlt(hMemoryDC, 40, 0, 24, 11, PATORDEST);
PatBlt(hMemoryDC, 24, 10, 24, 12, PATORDEST);
```

```
PatBlt(hMemoryDC, 0, 22, 24, 11, PATORDEST);

SelectObject(hMemoryDC, hOldBrush);
DeleteObject(hRedBrush);
DeleteObject(hGreenBrush);
DeleteObject(hBlueBrush);

SelectObject(hMemoryDC, hOldBitmap);
DeleteDC(hMemoryDC);
ReleaseDC(hWnd, hDC);
```

In this example, the **CreateSolidBrush** function creates the red, green, and blue brushes needed to make the plaid pattern. The **SelectObject** function selects each brush into the memory device context as that brush is needed, and the **PatBlt** function paints the colors into the bitmap. Each color is painted three times, each time into a small rectangle. **PatBlt** intentionally overlaps the different color rectangles a little. Since the PATORDEST raster-operation code is given, **PatBlt** combines the brush color with the color already in the bitmap by using a Boolean OR operator. The result is a different color border around each rectangle.

9.2.5 Deleting Bitmaps

A bitmap, like any resource, occupies memory while in use. After you have finished using a bitmap or before your application terminates, it is important that you delete the bitmaps you have created in order to make that memory available to other applications. To delete a bitmap, you first need to remove it from any device context it may currently be selected in, then you can delete it by using the **DeleteObject** function.

The following example deletes the bitmap identified by the hBitmap parameter, after removing it as the currently selected bitmap in the memory device context identified by the hMemoryDC parameter:

```
SelectObject(hMemoryDC, hOldBitmap);
DeleteObject(hBitmap);
```

The **SelectObject** function removes the bitmap from selection by replacing it with a previous bitmap identified by the hOldBitmap parameter. The **DeleteObject** function deletes the bitmap. Thereafter, the bitmap handle in the hBitmap parameter is no longer valid and must not be used.

9.3 Displaying Bitmaps

You can display a bitmap in the following three ways:

- Use the **BitBlt** function to copy the bitmap from a memory display context to a display device.
- Use the **StretchBlt** function to copy a stretched or compressed bitmap from a memory display context to a display device.
- Use the **CreatePatternBrush** function to create a brush that incorporates the bitmap. Any subsequent GDI functions that use the brush, such as **PatBlt**, will display the bitmap.

You can also display a bitmap in a menu. In such a case, the bitmap is used as a menu item that the user can choose to carry out an action. For details, see Chapter 8, "Menus."

9.3.1 Displaying a Bitmap

You can display any bitmap by using the **BitBlt** function. This function copies a bitmap from a source to a destination device context. To display a bitmap with **BitBlt**, you need to create a memory device context and select the bitmap into it first. The following example displays the bitmap by using **BitBlt**:

```
HDC hDC, hMemoryDC;
    .
    .
    .
hDC = GetDC(hWnd);
hMemoryDC = CreateCompatibleDC(hDC);

hOldBitmap = SelectObject(hMemoryDC, hBitmap);

BitBlt(hDC, 100, 30, 64, 32, hMemoryDC, 0, 0, SRCCOPY);

SelectObject(hMemoryDC, hOldBitmap);
DeleteDC(hMemoryDC);
ReleaseDC(hWnd, hDC);
```

The **GetDC** function specifies the display context for the client area of the window identified by the hWnd variable. The **CreateCompatibleDC** function creates a memory device context that is compatible with the display context. The **SelectObject** function selects the bitmap, identified by the hBitmap variable, into the memory device context and returns the previously selected bitmap. If **SelectObject** cannot select the bitmap, it returns zero.

The **BitBlt** function copies the bitmap from the memory device context to the display context. The function places the upper-left corner of the bitmap at the point (100,30). The entire bitmap, 64 bits wide by 32 bits high, is copied. The hDC and hMemoryDC variables identify the destination and source contexts, respectively. The constant, SRCCOPY, is the raster-operation code. It directs **BitBlt** to copy the source bitmap without combining it with patterns or colors already at the destination.

The **SelectObject**, **DeleteDC**, and **ReleaseDC** functions clean up after the bitmap has been displayed. In general, when you have finished using memory and display contexts, you should release them as soon as possible—especially display contexts, which are a limited resource. Windows maintains a cache of display contexts that all applications draw from. If an application does not release a display context after using it, other applications may not be able to retrieve a context when needed. The **SelectObject** function is required since you must not delete a device context while any bitmap other than the context's original bitmap is selected.

In the previous example, the width and height of the bitmap were assumed to be 64 and 32 pixels, respectively. Another way to specify the width and height of the bitmap to be displayed is to retrieve the width and height from the bitmap itself. You can do this by using the **GetObject** function, which fills a specified structure with the dimensions of the given object. For example, to retrieve the width and height of a bitmap, you would use the following statements:

```
BITMAP Bitmap;
    .
    .
    .
GetObject(hBitmap, (LPBITMAP) &Bitmap);
```

The next example copies the width and height of the bitmap to the **bmWidth** and **bmHeight** fields of the structure, Bitmap. You can use these values in **BitBlt** as follows:

```
BitBlt(hDC, 100, 30, Bitmap.bmWidth, Bitmap.bmHeight,
    hMemoryDC, 0, 0, SRCCOPY);
```

The **BitBlt** function can display both monochrome and color bitmaps. No special steps are required to display bitmaps of different formats. However, you should be aware that **BitBlt** may convert the bitmap if its color format is not the same as the destination device. For example, when displaying a color bitmap on a monochrome display, **BitBlt** converts the pixels having the current background color to white and all other pixels to black.

9.3.2 Adding Color to Monochrome Bitmaps

If your computer has a color display, you can add color to a monochrome bitmap by setting the foreground and background colors of the display context. The foreground and background colors specify which colors the white and black bits of the bitmap will have when displayed. You set the foreground and background colors by using the **SetTextColor** and **SetBkColor** functions. The following example shows how to set the foreground color to red and the background color to green:

```
SetTextColor(hDC, RGB(255,0,0));
SetBkColor(hDC, RGB(0,255,0));
```

The hDC variable holds the handle to the display context. The **SetTextColor** function sets the foreground color to red. The **SetBkColor** function sets the background color to green. The **RGB** utility creates an RGB color value by using the three specified values. Each value represents an intensity for each of the primary colors—red, green, and blue—with the value 255 representing the highest intensity, and zero, the lowest. You can produce colors other than red and green by combining the color intensities. For example, the following statement creates a yellow RGB value:

```
RGB(255,255,0)
```

Once the foreground and background colors are set, no further action is required. You can display a bitmap (as described earlier) and Windows will automatically add the foreground and background colors. The foreground color is applied to the white bits (the bits set to 1) and the background color to the black bits (the bits set to zero). Note that the background mode, as specified by the **SetBkMode** function, does not apply to bitmaps. Also, the foreground and background colors do not apply to color bitmaps.

When displayed in color, the bitmap named "dog" will be red, the background will be green.

9.3.3 Stretching Bitmaps

Your bitmaps are not limited to their original size. You can stretch or compress them by using the **StretchBlt** function in place of **BitBlt**. For example, you can double the size of a 64-by-32-pixel bitmap by using the following statement:

```
StretchBlt(hDC, 100, 30, 128, 64, hMemoryDC, 0, 0, 64, 32, SRCCOPY);
```

The **StretchBlt** function has two additional parameters that **BitBlt** does not. In particular, **StretchBlt** specifies the width and height of the source bitmap. The first width and height, given as 128 and 64 pixels in the

previous example, apply only to the final size of the bitmap on the destination device context.

To compress a bitmap, **StretchBlt** removes pixels from the copied bitmap. This means that some of the information in the bitmap is lost when it is displayed. To minimize the loss, you can set the current stretching mode to direct **StretchBlt** to attempt to save some of the information by combining it with the pixels that will be displayed. The stretching mode can be WHITEONBLACK, BLACKONWHITE, or COLORONCOLOR. You use WHITEONBLACK if you wish to preserve white pixels at the expense of black pixels; for example, if you have a white outline on a black background. You use BLACKONWHITE for just the opposite (a black outline on a white background). COLORONCOLOR is used for color bitmaps where attempting to combine colors can lead to undesirable effects.

The **SetStretchBltMode** function sets the stretching mode. In the following example, **SetStretchBltMode** sets the stretching mode to WHITEONBLACK:

```
SetStretchBltMode(hDC, WHITEONBLACK);
```

9.3.4 Using Bitmaps in a Pattern Brush

You can use bitmaps in a brush by creating a pattern brush. Once the pattern brush is created, you can select the brush into a device context and use the **PatBlt** function to copy it to the screen; or the **Rectangle**, **Ellipse**, and other drawing functions can use the brush to fill interiors. When Windows draws with a pattern brush, it fills the specified area by repeatedly copying the bitmap horizontally and vertically, as necessary. It does not adjust the size of the bitmap to fit in the area as the **StretchBlt** function does.

If you use a bitmap in a pattern brush, the bitmap should be at least eight pixels wide by eight pixels high—the default pattern size used by most display drivers. (You can use large bitmaps, but only the upper-left, 8-by-8 corner will be used.) You may hard-code the bitmap, create and draw it, or load it as a resource. In any case, once you have the bitmap handle, you can create the pattern brush by using the **CreatePatternBrush** function. The following example loads a bitmap and uses it to create a pattern brush:

```
hBitmap = LoadBitmap(hInstance, "checks");
hBrush = CreatePatternBrush(hBitmap);
```

Once a pattern brush is created, you can select it into a device context by using the **SelectObject** function:

```
hOldBrush = SelectObject(hDC, hBrush);
```

Since the bitmap is part of the brush, this call to the **SelectObject** function does not affect the device context's selected bitmap.

After selecting the brush, you can use the **PatBlt** function to fill a specified area with the bitmap. For example, the following statement fills the upper-left corner of a window with the bitmap:

```
PatBlt(hDC, 0, 0, 100, 100, PATCOPY);
```

The PATCOPY raster operation directs **PatBlt** to completely replace the destination image with the pattern brush.

You can also use a pattern brush as a window's background brush. To do this, simply assign the brush handle to the **hbrBackground** field of the window-class structure as in the following example:

```
pWndClass->hbrBackground = CreatePatternBrush(hBitmap);
```

Thereafter, Windows uses the pattern brush when it erases the window's background. You can also change the current background brush for a window class by using the the **SetClassWord** function. For example, if you want to use a new pattern brush after a window has been created, you can use the following statement:

```
SetClassWord(hWnd, GCW_HBRBACKGROUND, hBrush);
```

Be aware that this statement changes the background brush for all windows of this class. If you only want to change the background for one window, you need to explicitly process the WM_ERASEBKGND messages that window receives. The following example shows how to process this message:

```
RECT Rect;
HBRUSH hOldBrush;
    .
    .
    .
case WM_ERASEBKGND:
    UnrealizeObject(hMyBkgndBrush);
    hOldBrush = SelectObject(wParam, hMyBkgndBrush);
    GetUpdateRect(wParam, &Rect, FALSE);
    PatBlt(wParam, Rect.left, Rect.top,
        Rect.right - Rect.left, Rect.bottom - Rect.top,
        PATCOPY);
    SelectObject(wParam, hOldBrush);
    break;
```

The WM_ERASEBKGND message passes a handle to a display context in the wParam parameter. The **SelectObject** function selects the desired background brush into the display context. The **GetUpdateRect** function retrieves the area that needs to be erased (this is not always the entire client area). The **PatBlt** function copies the pattern, overwriting anything

already in the update rectangle. The final **SelectObject** function restores the previous brush to the display context.

The **UnrealizeObject** function is used in the preceding example. Whenever your application or the user moves a window in which you have used or will use a pattern brush, you need to align your pattern brushes to the new position by using the **UnrealizeObject** function. This function resets a brush's drawing origin so that any patterns output after the move match the patterns output before the move.

You can use the **DeleteObject** function to delete a pattern brush when it is no longer needed. This function does not, however, delete the bitmap along with the brush. To delete the bitmap, you need to use **DeleteObject** again and specify the bitmap handle.

9.4 A Sample Application: Bitmap

This sample shows how to incorporate a variety of bitmap operations in an application. In particular, it shows how to do the following:

- Load and display a monochrome bitmap.
- Create and display a color bitmap.
- Stretch and compress a bitmap using the mouse.
- Set the stretching mode.
- Create and use a pattern brush.
- Use a pattern brush for the window background.

In this application, the user specifies (by using the mouse) where and how the bitmap is to be displayed. If the user drags the mouse while holding down the left button, and then releases that button, the application uses the **StretchBlt** function to fill the selected rectangle with the current bitmap. If the user clicks the right button, the application uses the **BitBlt** function to display the bitmap.

To create the Bitmap application, copy and rename the source files for the Generic application, then make the following modifications:

1. Add constant definitions and a function declaration to the include file.
2. Add two monochrome bitmaps, created by using Icon Editor, to the resource script file.

3. Add Bitmap, Pattern, and Mode menus to the resource script file.
4. Add global variables.
5. Add the WM_CREATE case to the window function to create bitmaps and add bitmaps to the menus.
6. Modify the WM_DESTROY case in the window function to delete bitmaps.
7. Add the WM_LBUTTONUP, WM_MOUSEMOVE, and WM_LBUTTONDOWN cases to the window function to create a selection rectangle and display bitmaps.
8. Add the WM_RBUTTONUP case to the window function to display bitmaps.
9. Add the WM_ERASEBKGND case to the window function to erase the client area.
10. Add the WM_COMMAND case to support the menus.
11. Modify the **link4** command line in the **make** file to include the *select.lib* library file.
12. Compile and link the application.

The following sections explain each step in detail.

9.4.1 Modify the Include File

You need to add the following function declarations and constant definitions to the include file:

```
#define IDM_BITMAP1             200
#define IDM_BITMAP2             201
#define IDM_BITMAP3             202

#define IDM_PATTERN1            300
#define IDM_PATTERN2            301
#define IDM_PATTERN3            302
#define IDM_PATTERN4            303

#define IDM_BLACKONWHITE        400
#define IDM_WHITEONBLACK        401
#define IDM_COLORONCOLOR        402

#define PATORDEST       0x00FA0089L

HBITMAP MakeColorBitmap(HWND);
```

9.4.2 Add the Bitmap Resources

You need to add two **BITMAP** statements to your resource script file. The two statements add the bitmaps "dog" and "cat" to your application resources. Add the following statements:

```
dog BITMAP dog.bmp
cat BITMAP cat.bmp
```

The "dog" bitmap is the white outline of a dog on a black background. The "cat" bitmap is the black outline of a cat on a white background.

9.4.3 Add the Bitmap, Pattern, and Mode Menus

You need to add a **MENU** statement to your resource script file. This statement defines the Bitmap, Pattern, and Mode menus used to choose the various bitmaps and modes that are used in the application. Add the following statement to the resource script file:

```
bitmap MENU
BEGIN
    POPUP "&Bitmap"
    BEGIN
        MENUITEM "", IDM_BITMAP1
    END

    POPUP "&Pattern"
    BEGIN
        MENUITEM "", IDM_PATTERN1
    END

    POPUP "&Mode"
    BEGIN
        MENUITEM "&WhiteOnBlack", IDM_WHITEONBLACK, CHECKED
        MENUITEM "&BlackOnWhite", IDM_BLACKONWHITE
        MENUITEM "&ColorOnColor", IDM_COLORONCOLOR
    END
END
```

The Bitmap and Pattern menus each contain a single **MENUITEM** statement. This statement defines a command that serves as a placeholder only. The application will add the actual commands to use in the menu by using the **ChangeMenu** function.

9.4.4 Add Global Variables

You need to declare the pattern arrays, the bitmap handles and context handles, and other variables used to create and display the bitmaps. Add the following statements to the beginning of your source file:

```
short White[] =  { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };
short Black[] =  { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
short Zigzag[] = { 0xFF, 0xF7, 0xEB, 0xDD, 0xBE, 0x7F, 0xFF, 0xFF };
short CrossHatch[] = { 0xEF, 0xEF, 0xEF, 0xEF, 0x00, 0xEF, 0xEF, 0xEF };

HBITMAP hPattern1;
HBITMAP hPattern2;
HBITMAP hPattern3;
HBITMAP hPattern4;
HBITMAP hBitmap1;
HBITMAP hBitmap2;
HBITMAP hBitmap3;
HBITMAP hMenuBitmap1;
HBITMAP hMenuBitmap2;
HBITMAP hMenuBitmap3;
HBITMAP hBitmap;
HBITMAP hOldBitmap;

HBRUSH hBrush;                  /* brush handle                        */
int fStretchMode;               /* type of stretch mode to use         */

HDC hDC;                        /* handle to device context            */
HDC hMemoryDC;                  /* handle to memory device context     */
BITMAP Bitmap;                  /* bitmap structure                    */

BOOL bTrack = FALSE;            /* TRUE if user is selecting a region */
RECT Rect;

WORD wPrevBitmap = IDM_BITMAP1;
WORD wPrevPattern = IDM_PATTERN1;
WORD wPrevMode = IDM_WHITEONBLACK;
WORD wPrevItem;

int Shape = SL_BLOCK; /* shape to use for the selection rectangle */
```

The pattern arrays White, Black, Zigzag, and CrossHatch contain the bits defining the 8-by-8-pixel bitmap images. The variables hPattern1, hPattern2, hPattern3, and hPattern4 hold the bitmap handles of the brush patterns. The variables hBitmap1, hBitmap2, and hBitmap3 hold the bitmap handles of the bitmaps to be displayed. The variables hMenuBitmap1, hMenuBitmap2, and hMenuBitmap3 hold the bitmap handles of bitmaps to be displayed in the Bitmaps menu. The variables hBrush, hBitmap, and fStretchMode hold the current background brush, bitmap, and stretching mode. The variables hDC, hMemoryDC, and hOldBitmap hold handles used with the memory device context. The Bitmap structure holds the dimensions of the current bitmap. The bTrack variable is used to indicate a selection in progress. The Rect structure holds the current

selection rectangle. The variables wPrevBitmap, wPrevPattern, wPrevMode, and wPrevItem hold the menu IDs of the previously chosen bitmap, pattern, and stretching mode. These are used to place and remove checkmarks in the menus.

9.4.5 Add the WM_CREATE Case

You need a WM_CREATE case and supporting variable and function declarations to create or load the bitmaps and to set the menus. The WM_CREATE case creates four 8-by-8-pixel, monochrome bitmaps to be used as patterns in a pattern brush for the window background. It also creates or loads three 64-by-32-pixel bitmaps to be displayed in the window. To let the user choose a bitmap or pattern for viewing, the WM_CREATE case adds them to the Bitmap and Pattern menus by using the **ChangeMenu** function. Finally, the case sets the initial values of the brush, bitmap, and stretching modes and creates the memory device context from which the bitmaps are copied.

The WM_CREATE case creates the four patterns by using the **CreateBitmap** function. It loads two bitmaps, "dog" and "cat", and creates a third by using the **MakeColorBitmap** function defined within the application. Once the patterns and bitmaps have been created, the WM_CREATE case creates pop-up menus, appends the patterns and bitmaps to the menus, and replaces the existing Bitmap and Pattern menus with the new pop-ups. Next, the hBrush, hBitmap, and fStretchMode variables are set to the initial values for the background brush, bitmap, and stretching modes. Finally, the case creates the memory device context from which the bitmaps will be copied to the display. Add the following statements to your window function:

```
case WM_CREATE: /* message: create window */

    hPattern1 = CreateBitmap(8, 8, 1, 1, (LPSTR) White);
    hPattern2 = CreateBitmap(8, 8, 1, 1, (LPSTR) Black);
    hPattern3 = CreateBitmap(8, 8, 1, 1, (LPSTR) Zigzag);
    hPattern4 = CreateBitmap(8, 8, 1, 1, (LPSTR) CrossHatch);

    hBitmap1 = LoadBitmap(hInst, "dog");
    hBitmap2 = LoadBitmap(hInst, "cat");
    hBitmap3 = MakeColorBitmap(hWnd);

    hMenuBitmap1 = LoadBitmap(hInst, "dog");
    hMenuBitmap2 = LoadBitmap(hInst, "cat");
    hMenuBitmap3 = MakeColorBitmap(hWnd);

    hMenu = CreateMenu();
    ChangeMenu(hMenu, NULL, "&White", IDM_PATTERN1,
        MF_APPEND | MF_STRING | MF_CHECKED);
    ChangeMenu(hMenu, NULL, "&Black", IDM_PATTERN2,
        MF_APPEND | MF_STRING);
    ChangeMenu(hMenu, NULL, (LPSTR) (LONG) hPattern3, IDM_PATTERN3,
        MF_APPEND | MF_BITMAP);
    ChangeMenu(hMenu, NULL, (LPSTR) (LONG) hPattern4, IDM_PATTERN4,
        MF_APPEND | MF_BITMAP);
```

```
ChangeMenu(GetMenu(hWnd), 1, "&Pattern", hMenu,
    MF_CHANGE | MF_POPUP | MF_BYPOSITION);

hMenu = CreateMenu();

ChangeMenu(hMenu, NULL, (LPSTR) (LONG) hMenuBitmap1, IDM_BITMAP1,
    MF_APPEND | MF_BITMAP | MF_CHECKED);
ChangeMenu(hMenu, NULL, (LPSTR) (LONG) hMenuBitmap2, IDM_BITMAP2,
    MF_APPEND | MF_BITMAP);
ChangeMenu(hMenu, NULL, (LPSTR) (LONG) hMenuBitmap3, IDM_BITMAP3,
    MF_APPEND | MF_BITMAP);

ChangeMenu(GetMenu(hWnd), 0, "&Bitmap", hMenu,
    MF_CHANGE | MF_POPUP | MF_BYPOSITION);

hBrush = CreatePatternBrush(hPattern1);
fStretchMode = IDM_BLACKONWHITE;

hDC = GetDC(hWnd);
hMemoryDC = CreateCompatibleDC(hDC);
ReleaseDC(hWnd, hDC);
hOldBitmap = SelectObject(hMemoryDC, hBitmap1);
GetObject(hBitmap1, 16, (LPSTR) &Bitmap);

hMenu = GetSystemMenu(hWnd, FALSE);
ChangeMenu(hMenu, NULL, NULL, NULL, MF_APPEND | MF_SEPARATOR);
ChangeMenu(hMenu, NULL, "A&bout Bitmap...", ID_ABOUT,
    MF_APPEND | MF_STRING);
break;
```

The **CreateBitmap** and **LoadBitmap** functions work as described in earlier sections in this chapter. The **MakeColorBitmap** function is created for this application. It creates and draws a color bitmap, using the same method described in Section 9.2.2, "Creating and Drawing Bitmaps." The statements of this function are given later in this section. Notice that each bitmap is loaded or created twice. This is required since no single bitmap handle may be selected into two device contexts at the same time. To display in a menu requires a selection, and to display in the client area also requires a selection.

The **CreateMenu** function creates an empty menu and returns a handle to the menu. The **ChangeMenu** functions that specify the pattern handles add the patterns as menu items to the new menu. The MF_BITMAP option specifies that a bitmap is to be added. The **CheckMenuItem** function places a checkmark next to the current menu item, and the last **ChangeMenu** function replaces the existing Pattern menu. The same steps are then repeated for the Bitmap menu.

The **CreateCompatibleDC** function creates a memory device context that is compatible with the display. The **SelectObject** function selects the current bitmap into the memory device context so that it is ready to be copied to the display. The **GetObject** function copies the dimensions of the bitmap into the Bitmap structure. The structure can then be used in subsequent **BitBlt** and **StretchBlt** functions to specify the width and height of the bitmap.

The **MakeColorBitmap** function creates a color bitmap by creating a bitmap that is compatible with the display, then paints a plaid color pattern by using red, green, and blue brushes and the **PatBlt** function. Add the following function definition to the end of your source file:

```
HBITMAP MakeColorBitmap(hWnd)
HWND hWnd;
{
    HDC hDC;
    HDC hMemoryDC;
    HBITMAP hBitmap;
    HBITMAP hOldBitmap;
    HBRUSH hRedBrush;
    HBRUSH hGreenBrush;
    HBRUSH hBlueBrush;
    HBRUSH hOldBrush;

    hDC = GetDC(hWnd);
    hMemoryDC = CreateCompatibleDC(hDC);
    hBitmap = CreateCompatibleBitmap(hDC, 64, 32);
    hOldBitmap = SelectObject(hMemoryDC, hBitmap);
    hRedBrush = CreateSolidBrush(RGB(255,0,0));
    hGreenBrush = CreateSolidBrush(RGB(0,255,0));
    hBlueBrush = CreateSolidBrush(RGB(0,0,255));

    PatBlt(hMemoryDC, 0, 0, 64, 32, BLACKNESS);
    hOldBrush = SelectObject(hMemoryDC, hRedBrush);
    PatBlt(hMemoryDC, 0, 0, 24, 11, PATORDEST);
    PatBlt(hMemoryDC, 40, 10, 24, 12, PATORDEST);
    PatBlt(hMemoryDC, 20, 21, 24, 11, PATORDEST);
    SelectObject(hMemoryDC, hGreenBrush);
    PatBlt(hMemoryDC, 20, 0, 24, 11, PATORDEST);
    PatBlt(hMemoryDC, 0, 10, 24, 12, PATORDEST);
    PatBlt(hMemoryDC, 40, 21, 24, 11, PATORDEST);
    SelectObject(hMemoryDC, hBlueBrush);
    PatBlt(hMemoryDC, 40, 0, 24, 11, PATORDEST);
    PatBlt(hMemoryDC, 20, 10, 24, 12, PATORDEST);
    PatBlt(hMemoryDC, 0, 21, 24, 11, PATORDEST);

    SelectObject(hMemoryDC, hOldBrush);
    DeleteObject(hRedBrush);
    DeleteObject(hGreenBrush);
    DeleteObject(hBlueBrush);
    SelectObject(hMemoryDC, hOldBitmap);
    DeleteDC(hMemoryDC);
    ReleaseDC(hWnd, hDC);
    return (hBitmap);
}
```

This function carries out the same steps described at the end of Section 9.2.3, "Creating Bitmaps with Hard-Coded Bits."

9.4.6 Modify the WM_DESTROY Case

You need to delete the bitmaps, patterns, brushes, and memory device context you have created before terminating the application. You delete bitmaps, patterns, and brushes by using the **DeleteObject** function. You delete the memory device context by using the **DeleteDC** function. Modify the WM_DESTROY case so that it looks like this:

```
case WM_DESTROY: /* message: destroy window */
    SelectObject(hMemoryDC, hOldBitmap);
    DeleteDC(hMemoryDC);
    DeleteObject(hBrush);
    DeleteObject(hPattern1);
    DeleteObject(hPattern2);
    DeleteObject(hPattern3);
    DeleteObject(hPattern4);
    DeleteObject(hBitmap1);
    DeleteObject(hBitmap2);
    DeleteObject(hBitmap3);
    DeleteObject(hMenuBitmap1);
    DeleteObject(hMenuBitmap2);
    DeleteObject(hMenuBitmap3);

    PostQuitMessage(0);
    break;
```

9.4.7 Add WM_LBUTTONUP, WM_MOUSEMOVE, and WM_LBUTTONDOWN Cases

You need to add WM_LBUTTONUP, WM_MOUSEMOVE, and WM_LBUTTONDOWN cases to the window function to let the user select a rectangle in which to copy the current bitmap. These cases use the selection functions (described in Appendix C, "Windows Libraries") to create a selection rectangle and supply feedback to the user. The WM_LBUTTONUP case then uses the **StretchBlt** function to fill the rectangle. Add the following statements to your window function:

```
case WM_LBUTTONDOWN: /* message: left mouse button pressed */

    bTrack = TRUE;
    SetRectEmpty(&Rect);
    StartSelection(hWnd, MAKEPOINT(lParam), &Rect,
        (wParam & MK_SHIFT) ? (SL_EXTEND | Shape) : Shape);
    break;

case WM_MOUSEMOVE: /* message: mouse movement */

    if (bTrack)
        UpdateSelection(hWnd, MAKEPOINT(lParam), &Rect, Shape);
    break;
```

```
case WM_LBUTTONUP: /* message: left mouse button released */

    bTrack = FALSE;
    EndSelection(MAKEPOINT(lParam), &Rect);
    ClearSelection(hWnd, &Rect, Shape);

    hDC = GetDC(hWnd);
    SetStretchBltMode(hDC, fStretchMode);
    StretchBlt(hDC, Rect.left, Rect.top,
        Rect.right - Rect.left, Rect.bottom - Rect.top,
        hMemoryDC, 0, 0,
        Bitmap.bmWidth, Bitmap.bmHeight,
        SRCCOPY);
    ReleaseDC(hWnd, hDC);
    break;
```

To use these functions, you also must include the **select.h** file (defined in Appendix C, "Windows Libraries"). Add the following statement to the beginning of your source file:

```
#include "Select.h"
```

9.4.8 Add the WM_RBUTTONUP Case

You need to add a WM_RBUTTONUP case to display the current bitmap by using the **BitBlt** function. Add the following statements to your window function:

```
case WM_RBUTTONUP: /* message: right mouse button released */

    hDC = GetDC(hWnd);
    BitBlt(hDC, LOWORD(lParam), HIWORD(lParam),
        Bitmap.bmWidth, Bitmap.bmHeight,
        hMemoryDC, 0, 0, SRCCOPY);
    ReleaseDC(hWnd, hDC);
    break;
```

9.4.9 Add the WM_ERASEBKGND Case

You need to add a WM_ERASEBKGND case to make sure the selected background brush is used. Add the following statements to your window function:

```
case WM_ERASEBKGND: /* message: erase background */

    UnrealizeObject(hBrush);
    hOldBrush = SelectObject(wParam, hBrush);
    GetClientRect(hWnd, &Rect);
    PatBlt(wParam, Rect.left, Rect.top,
        Rect.right-Rect.left, Rect.bottom-Rect.top,
        PATCOPY);
    SelectObject(wParam, hOldBrush);
    return TRUE;
```

The hOldBrush variable is declared as a local variable. The **UnrealizeObject** function sets the pattern alignment if the window has moved. The **SelectObject** function sets the background brush and the **GetClientRect** function determines which part of the client area needs to be erased. The **PatBlt** function copies the pattern to the update rectangle. The final **SelectObject** function restores the previous brush.

9.4.10 Add the WM_COMMAND Case

You need to add a WM_COMMAND case to support the Bitmap, Pattern, and Mode menus. Add the following statements to your window function:

```
case WM_COMMAND: /* message: Windows command */
    switch (wParam) {

        case IDM_BITMAP1:
            wPrevItem = wPrevBitmap;
            wPrevBitmap = wParam;
            GetObject(hBitmap1, 16, (LPSTR) &Bitmap);
            SelectObject(hMemoryDC, hBitmap1);
            break;

        case IDM_BITMAP2:
            wPrevItem = wPrevBitmap;
            wPrevBitmap = wParam;
            GetObject(hBitmap2, 16, (LPSTR) &Bitmap);
            SelectObject(hMemoryDC, hBitmap2);
            break;

        case IDM_BITMAP3:
            wPrevItem = wPrevBitmap;
            wPrevBitmap = wParam;
            GetObject(hBitmap3, 16, (LPSTR) &Bitmap);
            hOurBitmap = SelectObject(hMemoryDC, hBitmap3);
            break;

        case IDM_PATTERN1:
            wPrevItem = wPrevPattern;
            wPrevPattern = wParam;
            DeleteObject(hBrush);
            hBrush = CreatePatternBrush(hPattern1);
            InvalidateRect(hWnd, (LPRECT) NULL, TRUE);
            UpdateWindow(hWnd);
            break;

        case IDM_PATTERN2:
            wPrevItem = wPrevPattern;
            wPrevPattern = wParam;
            DeleteObject(hBrush);
            hBrush = CreatePatternBrush(hPattern2);
            InvalidateRect(hWnd, (LPRECT) NULL, TRUE);
            UpdateWindow(hWnd);
            break;
```

```
        case IDM_PATTERN3:
            wPrevItem = wPrevPattern;
            wPrevPattern = wParam;
            DeleteObject(hBrush);
            hBrush = CreatePatternBrush(hPattern3);
            InvalidateRect(hWnd, (LPRECT) NULL, TRUE);
            UpdateWindow(hWnd);
            break;

        case IDM_PATTERN4:
            wPrevItem = wPrevPattern;
            wPrevPattern = wParam;
            DeleteObject(hBrush);
            hBrush = CreatePatternBrush(hPattern4);
            InvalidateRect(hWnd, (LPRECT) NULL, TRUE);
            UpdateWindow(hWnd);
            break;

        case IDM_BLACKONWHITE:
            wPrevItem = wPrevMode;
            wPrevMode = wParam;
            fStretchMode = BLACKONWHITE;
            break;

        case IDM_WHITEONBLACK:
            wPrevItem = wPrevMode;
            wPrevMode = wParam;
            fStretchMode = WHITEONBLACK;
            break;

        case IDM_COLORONCOLOR:
            wPrevItem = wPrevMode;
            wPrevMode = wParam;
            fStretchMode = COLORONCOLOR;
            break;
    }

    CheckMenuItem(GetMenu(hWnd), wPrevItem, MF_UNCHECKED);
    CheckMenuItem(GetMenu(hWnd), wParam, MF_CHECKED);
    break;
```

9.4.11 Modify the Make file

You need to modify the **link4** command line in the **make** file to include the *select.lib* library file. This file contains the import declarations for the selection routines that are used with the WM_LBUTTONUP, WM_MOUSEMOVE, and WM_LBUTTONDOWN cases. You create the library as described in Appendix C, "Window Libraries." The new **link4** command line should look like this:

```
link4 bitmap, , , slibw/NOE select.lib, bitmap.def
```

9.4.12 Compile and Link

After making the necessary changes, compile and link the Bitmap application. Start Windows and then the Bitmap application. If you drag the mouse, using the left button, to form a rectangle then release the button, the window should look like Figure 9.2:

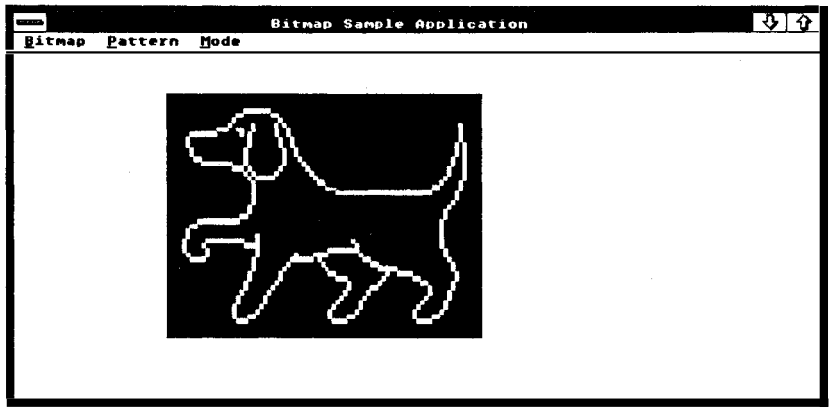


Figure 9.2 Bitmap Window with Dog

Use the menus to change the background and the stretching mode. Note the effect of the stretching mode on the "dog" and "cat" bitmaps.

Chapter 10
Controls and Dialog Boxes

10.1 Introduction

This chapter explains how to create controls and dialog boxes and how to use them in Microsoft Windows applications. Controls and dialog boxes are special windows that have features and capabilities that other windows do not. Controls and dialog boxes are designed to provide easy methods for interaction with the user.

10.2 Using Controls

A control is a predefined child window that carries out a specific kind of input or output. In Windows, controls are used as ready-made windows. For example, if you need a filename from the user to complete a task, you can create and display an edit control to let the user type the name.

A control, like any other window, belongs to a window class. The window class defines the default attributes of the control, but most importantly, defines the control window function. It is the window function that determines what the control will look like and how it will respond to user input. Control window functions are predefined in Windows, so no extra coding is required in your application when you use a control.

Windows has the following built-in control classes:

Class	Description
Button	Produces small, labeled windows that the user can choose to generate yes/no, on/off type of input.
Static	Produces small windows containing text or simple graphics. These are often used to label other controls or to separate a group of controls.
List box	Produces windows that contain lists of names from which the user can select one or more names.
Edit	Produces windows in which the user can enter and edit text.
Scroll bar	Produces windows that look and function like scroll bars in a window.

The following sections explain how to use these control classes to create and use controls in your application's windows.

10.2.1 Creating a Control

You can create a control by using the **CreateWindow** function. You use **CreateWindow** just as you would when creating a main window, but for a control, you need to specify the control class, the control style, the parent window, and the control ID. The following example shows how to create a push-button control:

```
hButtonWnd = CreateWindow("Button",          /* button class     */
    "OK",                                    /* button label     */
    BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE,
    20,                                      /* x-coordinate     */
    40,                                      /* y-coordinate     */
    30,                                      /* width in pixels  */
    12,                                      /* height in pixels */
    hWnd,                                    /* parent window    */
    IDOK,                                    /* control ID       */
    hInstance,                               /* instance handle  */
    NULL);
```

This example creates a push-button control that belongs to the "Button" window class and has the BS_PUSHBUTTON style. The control is a child window and will be visible when first created. The WS_CHILD style is required, but you do not need to specify the WS_VISIBLE style if you plan to use the **ShowWindow** function to show the control. **CreateWindow** places the control at the point (20,40) in the parent window's client area. The width and height are 40 and 12 pixels, respectively. The parent window is identified by the hWnd handle. The constant IDOK is the control identifier.

The **CreateWindow** function returns a handle to the control that you can use in subsequent functions to move, size, paint, or destroy a window, or to direct a window to carry out tasks.

10.2.2 Choosing a Control Style

The control styles, which depend on the control class, determine the appearance and function of the control. The following is a list of commonly used styles:

Style	Description
BS_PUSHBUTTON	Specifies a push-button control: a rounded rectangle containing a label that the user can choose in order to notify the parent window.
BS_DEFPUSHBUTTON	Specifies a default push button. A default push button is identical to a push button except that it has a heavier border.

BS_CHECKBOX	Specifies a check-box control. The user can select the box to turn the control on and off. When the control is on, the box contains an "X".
BS_RADIOBUTTON	Specifies a radio-button control. The user can select a circle to turn the control on and off. When the control is on, the circle contains a solid bullet.
ES_LEFT	Specifies a single-line, left-adjusted edit control.
ES_MULTILINE	Specifies a multiple-line edit control.
SS_LEFT	Specifies a left-adjusted, static text control.
SS_RIGHT	Specifies a right-adjusted, static text control.
LBS_STANDARD	Specifies a standard list box. A standard list box includes a scroll bar and notifies its parent window when the user makes a selection.

You can find a complete list of styles in the *Microsoft Windows Programmer's Reference.*

10.2.3 Setting the Parent Window

Since every control is a child window, a control requires a parent window. You specify the parent window when you create the control. As with any child window, a control is affected by changes to the parent window. For example, if Windows disables the parent window, it disables the control as well. If Windows paints, moves, or destroys the parent window, it also paints, moves, or destroys the control.

Although a control may be any size and moved to any position, it is restricted to the client area of the parent window. Windows clips the window if you move it outside the client area or make it bigger than the client area.

10.2.4 Choosing a Control ID

When you create a control, you can give it a unique identifier, or control ID. You specify the control ID in the **CreateWindow** function in place of a menu handle. Controls cannot have menus. The control will supply the control ID in any notification messages it sends to the window function of the parent window. The control ID is especially useful if you have several controls in a window. It is the quickest, easiest way to distinguish one control from another.

10.2.5 Receiving Notification Messages

As the user interacts with the control, the control sends information about that interaction, in the form of a notification message, to the parent window. A notification message is a WM_COMMAND message in which the wParam parameter contains the control ID, and the lParam parameter contains the notification code and the control handle. For example, when the user types a letter in an edit control, the control sends a WM_COMMAND message containing the EN_CHANGED notification code to the window function of the parent.

Since a notification message has the same basic form as menu input, you can process notification messages much as you would menu input. If you have carefully selected control IDs that do not conflict with menu IDs, you can process notification messages in the same **switch** statement you use to process menu input.

10.2.6 Moving and Sizing a Control

You can move or size a control by using the **MoveWindow** function. This function moves the control to the specified point in the parent window's client area and sets the control to the given width and height. The following example moves a control to the point (10,10) in the client area and sets the width and height to 30 and 12 pixels, respectively.

```
MoveWindow(hButtonWnd, 10,10, 30,12);
```

Windows automatically moves a control when it moves the parent window. A control's position is always relative to the upper-left corner of the parent's client area, so when the parent moves, the control remains fixed in the client area but moves relative to the display. Windows does not size a control when it sizes the parent window, but it does send a WM_SIZE message to the parent to indicate the new size of the parent window. You can use this message to give the control a new size, if desired.

10.2.7 Sending Control Messages

Most controls accept and process a variety of control messages. These are special messages that direct a control to carry out some task that is unique to the control. For example, the EM_GETTEXTLENGTH message directs an edit control to return the length of a selected line of text.

You send a control message by using the **SendMessage** function. You must supply the message number and any required wParam and lParam parameter values. For example, the following statement sends the EM_GETTEXTLENGTH message to the edit control identified by the

handle hEditWnd; it then returns the length of the selected line in the edit control:

```
nLength = SendMessage(hEditWnd, EM_GETTEXTLENGTH, 0, 0L);
```

Many controls also process standard window messages, such as WM_HSCROLL and WM_VSCROLL. You can send these messages to controls by the same method you use to send control messages.

10.2.8 Enabling or Disabling Input to a Control

You can enable or disable input to a control by using the **EnableWindow** function. This function allows or prevents a control from receiving user input. The following example shows how to disable a control:

```
EnableWindow(hButton, FALSE);
```

You can restore input to the control by enabling it using the following function:

```
EnableWindow(hButton, TRUE);
```

10.2.9 Destroying a Control

You can destroy a control by using the **DestroyWindow** function. This function deletes any internal record of the control and removes the control from the parent window's client area. The following example shows how to destroy a control:

```
DestroyWindow(hEditWnd);
```

Windows automatically destroys a control when it destroys the parent window. In general, you will need to destroy a window only if you no longer need it in the parent window.

10.3 Using Button Controls

A button control is a small window used for simple yes/no, on/off type of input. There are the following button-control styles:

- Push button
- Default push button
- Radio button

- Check box
- Group box

You can create a button control by using the "Button" class and specifying a button style. For example, the following call to the **CreateWindow** function creates a default push-button control with the label "OK":

```
HWND hDefButton;
    .
    .
    .
hDefButton = CreateWindow("Button", "OK",
    BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE,
    20,40, 30,12, hWnd, IDOK, hInstance, NULL);
```

In this example, the WS_VISIBLE style is specified, so the control is displayed when it is created. The control ID is IDOK. This constant is defined in the *windows.h* file and is intended to be used with default push buttons, such as this "OK" button.

A default push button is typically used to let the user signal the completion of some activity, such as filling in an edit control with a filename. A default push-button control, as with other button controls, responds to both mouse and keyboard input. If the user moves the mouse cursor into the control and clicks it, the button sends a BN_CLICKED notification message to the parent window. The button does not have to have the input focus to respond to mouse input. It does, however, require the focus to respond to keyboard input. To let the user use the keyboard, you must give the input focus to the button by using the **SetFocus** function. The user can then press the SPACEBAR to direct the button to send a BN_CLICKED notification message to the parent window.

A check box is typically used to select some option to use in the current task. For example, you might use a check box to let the user choose an italic font for the next output operation. You can create a check-box control by using the BS_CHECKBOX style, as in the following example:

```
# define ID_ITALIC    201
HWND hCheckBox;
    .
    .
    .

hCheckBox = CreateWindow("Button", "Italic",
    BS_CHECKBOX | WS_CHILD | WS_VISIBLE,
    20,40, 30,12, hWnd, ID_ITALIC, hInstance, NULL);
```

In this example, the check-box label is "Italic" and the control ID is ID_ITALIC.

A check box responds to mouse and keyboard input much as a push-button control would. That is, it sends a notification message to the parent window when the user clicks the control or presses the SPACEBAR. However, a check box can display a check (an "X") in its box to show that it is currently on (it has been selected). You can direct the control to show the check by using the BM_SETCHECK message. You can also test to see if the check box has a check by using the BM_GETCHECK message. For example, to place a check in the check box, use the following function:

```
SendMessage(hCheckBox, BM_SETCHECK, 1, 0L);
```

This means you can place or remove a check in the check box any time you want; for example, when the parent window function receives a BN_CLICKED notification message. Windows also provides a BS_AUTOCHECKBOX style that automatically places or removes a check.

Radio-button controls work in much the same way as check boxes. However, radio buttons are usually used in groups and represent mutually exclusive options. Group boxes are used to enclose two or more related radio buttons. Group boxes do not respond to user input; that is, they do not generate notification messages.

10.4 Using Static Controls

A static control is a small window that contains text or graphics. You typically use a static control to label some other control or to create boxes and lines that separate one group of controls from another.

The most commonly used static control is the SS_LEFT style. This is a left-adjusted line of text. That is, the control writes the line's text starting at the left end of the control, displaying as much of the label as will fit in the control and clipping the rest. The control uses the system font for the text, so you can compute an appropriate size for the control by retrieving the font metrics for this font (see Appendix A, "Fonts," for details).

Like group boxes, static controls do not respond to user input; that is, they do not generate notification messages when chosen. However, you can change the appearance and location of a static control at any time. For example, you can change the text associated with a static control by using the **SetWindowText** function or the WM_SETTEXT message.

10.5 Using List Boxes

A list box is a control for a list of character strings, such as filenames. You typically use a list box to display a list of names from which the user can select one or more. There are several styles associated with a list box. The most common styles are LBS_NOTIFY and LBS_SORT. These styles create a list box that sorts its names alphabetically and sends notification messages to the parent window when the user selects a name. The WS_VSCROLL style is often specified with a list box to create a scroll bar so that the user can scroll the list-box contents. These three styles are included in the LBS_STANDARD style.

You can add a string to a list box by using the LB_ADDSTRING message. This message copies the given string to the list box, which displays it in the list. If the list box has the LBS_SORT style, the string is sorted alphabetically. The following example shows how to add a string:

```
int nIndex;
    .
    .
    .
nIndex = SendMessage(hListBox, LB_ADDSTRING, NULL, (LPSTR) "Alfred");
```

This message returns an integer that represents the index of the string in the list. You can use this index in subsequent list-box messages to identify the string, but only as long as you do not add, delete, or insert any other string. Doing so may change the string's index. For example, you can delete the string from the list box by supplying the index with the LB_DELETESTRING message, as in the following example:

```
SendMessage(hListBox, LB_DELETESTRING, nIndex, NULL);
```

A list box responds to both mouse and keyboard input. If the user clicks a string or presses the SPACEBAR in the list box, the list box selects the string and indicates the selection by inverting the string text. If the list box has the LBS_NOTIFY style, the list box also sends an LBN_SELCHANGE notification message to the parent window. If the user double-clicks a string and LBS_NOTIFY is specified, the list box sends the messages LBN_SELCHANGE and LBN_DBLCLK to the parent window.

You can always retrieve the index of the selected string by using the LB_GETCURSEL and LB_GETTEXT messages. The LB_GETCURSEL message retrieves the selection's index in the list box, and the LB_GETTEXT message retrieves the selection from the list box, copying it to a buffer that you supply.

10.6 Using Edit Controls

An edit control is a rectangular child window in which the user can enter and edit text. Edit controls have a variety of features, such as multiple-line editing and scrolling. You specify the features you want by specifying a control style.

Edit control styles, like window styles, define how the control will look and operate. For example, the ES_MULTILINE style creates an edit control in which you can enter more than one line of text. The ES_AUTOHSCROLL and ES_AUTOVSCROLL styles direct the edit control to scroll horizontally or vertically if the user enters more text than can fit in the control's client area. If these styles are not specified and the user enters more text than can fit on one line, the control wraps to the next line.

An edit control sends notification messages to its parent window. For example, an edit control sends an EN_CHANGE message when the user makes a change to the text. An edit control can also receive messages, such as EM_GETLINE and EM_LINELENGTH. An edit control carries out the specified action when it receives a message.

10.7 Using Scroll Bars

Scroll bars are predefined windows that can be positioned anywhere in a window. They are used to provide scrolling input for the window. The scroll bar sends a notification message to its parent window whenever the user clicks the control with the mouse; this allows the parent window to process the messages so that proper scrolling can occur.

10.8 Designing Your Own Controls

You can design and use your own controls by creating a control class and writing the control window function. Controls are a convenient way to develop windows that provide special-purpose input or output, such as gauges, thermometers, and charts. Once you have designed and tested a control, you can use it in a variety of applications.

10.9 A Sample Application: EditCntl

This sample application illustrates how you can use an edit control in an application's main window to provide multiple-line text entry and editing. The EditCntl application fills the client area of its main window with a multiple-line edit control and monitors the size of the client area to ensure that the edit control always just fits.

To create the application, copy and rename the source files of the Edit-Menu application, then make the following modifications:

1. Add a new constant to the include file.
2. Add new variables.
3. Add a **CreateWindow** function.
4. Add a WM_SIZE case.
5. Compile and link the application.

10.9.1 Add a Constant to the Include File

You need to add a constant to the include file to serve as the Control ID for the edit control. Add the following statement:

```
#define ID.EDIT 300
```

10.9.2 Add New Variables

You need a global variable to hold the window handle of the edit control. Add the following statement to the beginning of the C-language source file:

```
HWND hEditWnd;                          /* handle to edit window */
```

You also need a local variable in the **WinMain** function to hold the coordinates of the client-area rectangle. These coordinates are used to determine the size of the control. Add the following statement to the beginning of the **WinMain** function:

```
RECT Rect;
```

10.9.3 Add a CreateWindow Function

You need to create the edit control by using the **CreateWindow** function, but before doing so, you need to retrieve the dimensions of the client area so that you can set the size of the control. Add the following statements to the **WinMain** function immediately after creating the main window:

```
GetClientRect(hWnd, (LPRECT) &Rect);

hEditWnd = CreateWindow("Edit",
    NULL,
    WS_CHILD | WS_VISIBLE |
    ES_MULTILINE |
    WS_VSCROLL | WS_HSCROLL |
    ES_AUTOHSCROLL | ES_AUTOVSCROLL,
    0,
    0,
    (Rect.right-Rect.left),
    (Rect.bottom-Rect.top),
    hWnd,
    ID_EDIT,
    hInst,
    NULL);

if (!hEditWnd) {
   DestroyWindow(hWnd);
   return (NULL);
}
```

The **GetClientRect** function retrieves the dimensions of the the main window's client area and places that information in the Rect structure. The **CreateWindow** function creates the edit control, using the width and height computed by the Rect structure.

The **CreateWindow** function creates the edit window. To create an edit control, you need to use the predefined "Edit" control class and you need to specify the WS_ CHILD window style. The predefined controls may be used as child windows only. They cannot be used as main or pop-up windows. Since a child window requires a parent window, the handle of the main window, hWnd, is specified in the function call.

For this edit control, a number of edit-control styles are also specified. Edit-control styles, like window styles, define how the control will look and operate. This edit control is a multiple-line control, meaning the user will be able to enter more than one line of text in the control window. Also, the control will automatically scroll horizontally or vertically if the user types more text than can fit in the window.

The upper-left corner of the edit control is placed at the upper-left corner of the parent window's client area. A child window's coordinates are always relative to the parent window's client area. The next two arguments, Rect.right–Rect.left and Rect.bottom–Rect.top, define the height

and width of the edit control, ensuring that the edit control fills the client area when the window is first displayed.

Since an edit control sends notification messages to its parent window, the control must be given a control ID. Child windows cannot have menus, so the menu argument in the **CreateWindow** function is used to specify the control ID instead. For this edit control, the ID is set to ID_EDIT. Any notification messages sent to the parent window by the edit control will contain this ID.

If the edit control cannot be created, the **CreateWindow** function returns NULL. In this case, the application cannot continue, so the **DestroyWindow** function is used to destroy the main window before terminating the application.

10.9.4 Add a WM_SIZE Case

You need to add a WM_SIZE case to the window function. Windows sends a WM_SIZE message to the window function whenever the width or height of a window changes. Since changing the main window size does not automatically change the edit-control size, the WM_SIZE case is needed to change the size of the control. Add the following statements to the window function:

```
case WM_SIZE:
    MoveWindow(hEditWnd, 0, 0, LOWORD(lParam),
        HIWORD(lParam), FALSE);
    break;
```

10.9.5 Compile and Link

No changes are required to the **make** file. Compile and link the EditCntl application, then start Windows and run the application. Now, you can insert text, backspace to delete text, and you can use the mouse instead of the keyboard to select text. And since you specified ES_MULTILINE, ES_AUTOVSCROLL, and ES_AUTOHSCROLL when creating the control, the control can edit a full screen of text, then scroll and edit more.

The EditCntl application illustrates the first step required to make a simple text editor. To make a complete editor, you can add a File menu to the main window to open and save text files and to copy or retrieve text from the edit control, and add an Edit menu to the main window to copy, cut, and paste text through the clipboard. Later chapters illustrate some simple ways to incorporate these features into your application.

10.10 What Is a Dialog Box?

A dialog box is a pop-up window that an application uses to display or prompt for information. Dialog boxes are typically used with commands to prompt the user for the information needed to complete a command. A dialog box contains one or more controls with which the user can enter text, choose options, and direct the action of a particular command.

You have already seen a dialog box in the Generic application: the About dialog box. This dialog box contains text controls that provide information about the application, and a push-button control that the user can use to close the dialog box and return to the main window. To process a dialog box, you need to supply a dialog-box template, a dialog function, and some means to call up the dialog box.

A dialog-box template is a description of the dialog box and the controls it contains. You create the template by using a text editor or the Windows 2.0 Dialog Editor, then add the template to your resource script file.

A dialog function is a callback function that Windows calls when it has messages for the dialog box. Although a dialog function is similar to a window function, Windows carries out special processing for dialog boxes, so the dialog function does not have the same responsibilities as a window function.

The most common way to call up a dialog box is to do so in response to menu input. For example, the Open and Save As commands in the File menu both require additional information to complete their tasks. They both display dialog boxes to prompt for the additional information.

To create a dialog box, you must follow these steps:

1. Create a dialog-box template and add it to the resource script file.
2. Create a dialog function to support the box.
3. Export the dialog function.

You call up a dialog box by using the **DialogBox** or **CreateDialog** function. These functions start modal and modeless dialog boxes, respectively.

10.10.1 Creating a Modal Dialog Box

You have already seen a modal dialog box (About) in the Generic application. A modal dialog box is a pop-up window that displays information and prompts for user input. It is called modal because it temporarily disables the parent window and forces the user to complete the requested action before returning control to the parent window.

Although you can give a modal dialog box almost any window style, the recommended styles are WS_DLGFRAME and WS_POPUP. The WS_DLGFRAME style gives the dialog box its characteristic double-line border. You should not use a menu, title bar, system menu, or other window feature common to overlapping windows. Since a modal dialog box does not have a system menu, you must supply at least one button that gives the user some way to terminate the box. For example, the About dialog box has a push button labeled "OK".

A modal dialog box starts its own message loop to process messages from the application queue without returning to the **WinMain** function. To keep input from going to the parent window, the dialog box disables the parent window before processing input. For this reason, a modal dialog box must never be created using the WS_CHILD style, since disabling the parent window also disables all child-style windows belonging to the parent.

You terminate a modal dialog box by using the **EndDialog** function.

10.10.2 Creating a Modeless Dialog Box

A modeless dialog box is simply a pop-up window that displays information or prompts for input from the user. Unlike a modal dialog box, a modeless dialog box does not disable the parent window. This means you can continue to work in the parent window while the modeless dialog box is displayed.

Most modeless dialog boxes have the WS_POPUP, WS_CAPTION, WS_BORDER, and WS_SYSTEMMENU styles. The typical modeless dialog box has a system menu, a title bar, and a thin black border. Although Windows automatically disables some of the system-menu commands for the dialog box, the menu still contains a Close command. The user can use this command instead of a push button to terminate the dialog box. You can also include controls in the dialog box, such as edit controls and check boxes.

A modeless dialog box receives its input through the message loop in the **WinMain** function. If you have controls in the dialog box and want to let the user move to and select controls by using the keyboard, you need to call the **IsDialogMessage** function in the main message loop. This function determines whether a keyboard input message is for the dialog box and, if necessary, processes it. The message loop for an application that has a modeless dialog box will look like this:

```
while (GetMessage(&msg, NULL, NULL, NULL) {
    if (hDlg == NULL || !IsDialogMessage(hDlg, &msg)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Since a modeless dialog box may not be present at all times, you need to check the hDlg variable that holds the handle in order to see if it is valid. If it is valid, **IsDialogMessage** determines whether the message is for the dialog box. If so, the message is processed and must not be further processed by using the **TranslateMessage** and **DispatchMessage** functions.

You terminate a modeless dialog box by using the **DestroyWindow** function, not the **EndDialog** function. **EndDialog** is used for modal dialog boxes only.

10.10.3 Creating a Dialog Function

A dialog function has the following form:

```
BOOL FAR PASCAL DlgFunc(hDlg, message, wParam, lParam)
HWND hDlg;
unsigned message;
WORD wParam;
DWORD lParam;
{
    switch (message) {

       /* Place message cases here */

       default:
            return FALSE;
   }
}
```

This is basically a window function, except that the **DefWindowProc** function is not called. Default processing of dialog-box messages is handled internally, so the dialog function must not call the **DefWindowProc** function.

The dialog function must be defined as a **FAR PASCAL** procedure, and must have the parameters given here. **BOOL** is the required return type.

Just as it does with window functions, Windows sends messages to a dialog function when it has information to give the function or wants the function to carry out some action. Unlike a window function, a dialog function responds to a message by returning a Boolean value. If the function processes the message, it returns TRUE. Otherwise, it returns FALSE.

In this function, the hDlg variable receives the handle of the dialog box. The other parameters serve the same purpose as in a window function. The **switch** statement is used as a filter for different messages. Most dialog functions process the WM_INITDIALOG and WM_COMMAND messages, but very little else.

The WM_INITDIALOG message, sent to the dialog box just before it is displayed, gives the dialog function the opportunity to give the input focus to any control in the dialog box. If the function returns TRUE, Windows will set the input focus to the control of its choosing. Since there is only one control in this dialog box, the dialog function lets Windows set the input focus.

The WM_COMMAND message is sent to the dialog function by the controls in the dialog box. If there are controls in the dialog box, they send notification messages when the user carries out some action within them. For example, a dialog function with a push button can check WM_COMMAND messages for the control ID of the push button. The control ID is in the wParam parameter. When it finds the ID, the dialog function can carry out the corresponding task.

10.10.4 Using Controls in Dialog Boxes

You use controls in dialog boxes much as you use them in regular windows. When a control is in a dialog box, however, you can use several special functions to access the control and send messages to it. For example, the **SendDlgItemMessage** function sends a message to a control in the dialog box, and the **SetDlgItemText** function sets the text of a control. You do not need to supply the control handle in these functions. Instead, you supply the dialog handle and the control ID. If you want the control handle, you can use the **GetDlgItem** function.

10.11 A Sample Application: FileOpen

This sample application shows how to build and use a modal dialog box to support the Open command in the File menu. The purpose and operation of the dialog box is fully described in the *Microsoft Windows Application Style Guide*. The FileOpen dialog box contains the following controls:

- A default push-button control, labeled "Open", used to direct the application to open the selected file.
- A button control, labeled "Cancel", used to cancel the Open command.
- A single-line edit control in which the user can enter the name of the file to open.

- A list box containing the names of files in the current directory from which the user can select the file to be opened. The list box also contains directory and drive names, which can be used to change the current directory or drive.
- Several static text controls used to label the list box and edit control, and to display the current directory name.

To create the FileOpen application, copy and rename the source files for the EditCntl application, then make the following modifications:

1. Add new constants to the include file.
2. Create the Open dialog-box template and add it to the resource script file.
3. Add new variables.
4. Add an IDM_OPEN case to the WM_COMMAND case.
5. Create the OpenDlg dialog function.
6. Add helper functions to support the OpenDlg dialog function.
7. Export the OpenDlg dialog function.
8. Compile and link the application.

10.11.1 Add Constants to the Include File

You need several new constants in the include file to identify the controls of the FileOpen dialog box. Add the following statements:

```
#define     ID_FILENAME   400
#define     ID_EDIT       401
#define     ID_FILES      402
#define     ID_PATH       403
#define     ID_LISTBOX    404
```

Although you may choose any integer for a control ID, the ID for each control in a given dialog box must be unique. Typically, a predefined ID, such as IDOK or IDCANCEL, is less than 100, so any number greater than 100 can be used for other controls.

10.11.2 Create the Open Dialog-Box Template

You need a dialog-box template in your resource script file to define the size and appearance of the Open dialog box. The **DIALOG** statement specifies the name and dimensions of a dialog box, as well as the controls the dialog box contains. Add the following statements:

```
Open DIALOG 10, 10, 148, 112
STYLE WS_DLGFRAME | WS_POPUP
BEGIN
    LTEXT "Open File &Name:", ID_FILENAME,  4,  4,  60, 10
    EDITTEXT ID_EDIT,                       4, 16, 100, 12, ES_AUTOHSCROLL
    LTEXT "&Files in", ID_FILES,            4, 40,  32, 10
    LISTBOX, ID_LISTBOX,                    4, 52,  70, 56, WS_TABSTOP
    LTEXT "",  ID_PATH,                    40, 40, 100, 10
    DEFPUSHBUTTON "&Open" , IDOK,          87, 60,  50, 14
    PUSHBUTTON "Cancel", IDCANCEL,         87, 80,  50, 14
END
```

The dialog box has a width and height (in dialog units) of 148 and 112, respectively. Dialog units are fractions of the default system-font character size and are used with dialog boxes to ensure that a dialog box has the same relative size, no matter which computer it is displayed on. The **BEGIN** and **END** statements are required.

The **DEFPUSHBUTTON** statement creates a default push button that is labeled "Open" and has the control ID, IDOK. In modal dialog boxes, pressing the ENTER key generates a notification message that uses the same ID, so you can permit the user to click the button or press ENTER to open the selected file.

The **PUSHBUTTON** statement creates the "Cancel" push button. Its ID is IDCANCEL, a predefined ID found in the *windows.h* file. In modal dialog boxes, pressing the ESCAPE key generates a notification message by using the same ID, so you can permit the user to click the button or press the ESCAPE key to cancel the Open command.

The first **LTEXT** statement creates a left-adjusted static control that contains the string, "Open File &Name:". This string serves as the label to the list box. In some dialog boxes, all static controls have this same ID. Although the general rule is to have a unique ID for each control in a dialog box, it is acceptable to use –1 for static controls, as long as the dialog function does not need to distinguish between them (for example, as long as the dialog function does not attempt to change the static-control text or position).

The **EDITTEXT** statement adds an edit control to the dialog box and identifies it with ID_EDIT. The ES_AUTOHSCROLL style is given so that the user can enter filenames that are longer than the control is wide.

The **LISTBOX** statement creates a list box. The ID of the list box is ID_LISTBOX. The width and height (in dialog units) of the list box are 70 and 56, respectively. The WS_TABSTOP style is given so that the user can move the focus to the list box by using the keyboard. Without this style, the only method the user has to access the list box is to click it with the mouse.

The last **LTEXT** statement creates a left-adjusted static control used to display the current directory and drive. The control is initially empty; the pathname is added later. This control also has a unique control ID, ID_PATH, to distinguish it from other static controls. This is important since you will use the **DlgDirList** function to fill the control.

10.11.3 Add New Variables

You need to declare several new global and local variables in order to hold the filename and the various pieces used to build the filename. Add the following statements at the beginning of your source file:

```
char FileName[128];                 /* current filename          */
char PathName[128];                 /* current pathname          */
char OpenName[128];                 /* filename to open          */
char DefPath[128];                  /* default path for list box */
char DefSpec[13] = "*.*";           /* default search spec       */
char DefExt[] = ".txt";             /* default extension         */
char str[255];                      /* string for sprintf() calls */
```

You need a new local variable to hold the procedure-instance address of the FileOpen dialog box. Add the following statement at the beginning of the window function:

```
FARPROC lpOpenDlg;
```

10.11.4 Add the IDM_OPEN Case

You need to fill in the IDM_OPEN case for the WM_COMMAND message, and you need to display the Open dialog box when the user chooses the command. Add the following statements to the window function:

```
case IDM_OPEN:
    lpOpenDlg = MakeProcInstance((FARPROC) OpenDlg, hInst));
    DialogBox(hInst, "Open", hWnd, lpOpenDlg);
    FreeProcInstance(lpOpenDlg);
    break;
```

The **MakeProcInstance** function creates a procedure-instance address for the OpenDlg function. The function ensures that the data segment for the current instance is used when the dialog function is called. Functions, such as OpenDlg, that are exported by an application may only be called through a procedure-instance address and must not be called directly.

The **FreeProcInstance** function is used to free a procedure-instance address when it is no longer needed. After the **DialogBox** function returns, the procedure-instance address, lpOpenDlg, is not needed and can be freed. It will be recreated the next time the dialog box is invoked.

The **DialogBox** function returns only after the dialog function has terminated the dialog box. This means the dialog box will complete any actions the user requests, before the application can continue execution. Such a dialog box is called a modal dialog box, since while it remains on the screen, the application is in a new mode of operation. This means the user can respond only to the dialog box. It also means that commands that apply to the application are not available while the dialog box is present.

10.11.5 Create the OpenDlg Function

You need to create an Open dialog box to process the various controls. When the dialog box is first displayed, the dialog function needs to fill the list box and the edit control, then give the input focus to the edit control and select the entire specification. If the user selects a filename in the list-box, the dialog function should copy the name to the edit control. If the user clicks the Open button, the dialog function should retrieve the filename from the edit control and prepare to open the file. If the user double-clicks a filename in the list box, the dialog function should retrieve the filename, copy it to the edit control, and prepare to open the file.

Add the following function to your source file:

```
HANDLE FAR PASCAL OpenDlg(hDlg, message, wParam, lParam)
HWND hDlg;
unsigned message;
WORD wParam;
LONG lParam;
{
    WORD index;          /* index to the filenames in the list box */
    PSTR pTptr;          /* temporary pointer                      */
    HANDLE hFile;        /* handle to the opened file              */

    switch (message) {
        case WM_COMMAND:
            switch (wParam) {
                case ID_LISTBOX:
                    switch (HIWORD(lParam)) {
                        case LBN_SELCHANGE:
                            if (!DlgDirSelect(hDlg, str, ID_LISTBOX)) {
                                SetDlgItemText(hDlg, ID_EDIT, str);
                                SendDlgItemMessage(hDlg, ID_EDIT,
                                    EM_SETSEL,
                                    NULL,
                                    MAKELONG(0, 0x7fff));
                            }
                            else {
                                strcat(str, DefSpec);
                                DlgDirList(hDlg, str, ID_LISTBOX,
                                    ID_PATH, 0x4010);
                            }
                            break;
                        case LBN_DBLCLK:
                            goto openfile;
                    }                       /* Ends ID_LISTBOX case */
                    return (TRUE);
```

```
                case IDOK:
openfile:
                    GetDlgItemText(hDlg, ID_EDIT, OpenName, 128);
                    if (strchr(OpenName, '*') ||
                        strchr(OpenName, '?')) {
                        SeparateFile(hDlg, (LPSTR) str, (LPSTR) DefSpec,
                            (LPSTR) OpenName);
                        if (str[0])
                            strcpy(DefPath, str);
                        ChangeDefExt(DefExt, DefSpec);
                        UpdateListBox(hDlg);
                        return (TRUE);
                    }
                    if (!OpenName[0]) {
                        MessageBox(hDlg, "No filename specified.",
                            NULL, MB_OK | MB_ICONQUESTION);
                        return (TRUE);
                    }
                    AddExt(OpenName, DefExt);
                    EndDialog(hDlg, NULL);
                    return(TRUE);

                case IDCANCEL:
                    EndDialog(hDlg, NULL);
                    return(TRUE);
            }
            break;

        case WM_INITDIALOG:                 /* Request to initalize   */
            UpdateListBox(hDlg);
            SetDlgItemText(hDlg, ID_EDIT, DefSpec);
            SendDlgItemMessage(hDlg,        /* dialog handle          */
                ID_EDIT,                    /* where to send message  */
                EM_SETSEL,                  /* select characters      */
                NULL,                       /* additional information */
                MAKELONG(0, 0x7fff));       /* Accept entire contents */
            SetFocus(GetDlgItem(hDlg, ID_EDIT));
            return (FALSE); /* Indicates focus is set to a control */
    }
    return (FALSE);
}
```

When the dialog function receives the WM_INITDIALOG message, the **SetDlgItemText** function copies the initial filename to the edit control, and the **SendDlgItemMessage** function sends the EM_SETSEL message to the control in order to select the entire contents of the edit control for editing. The **SetFocus** function gives the input focus to the edit control. (The **GetDlgItem** function retrieves the window handle of the edit control.) The UpdateListBox function, given at the beginning of the WM_INITDIALOG case, is a locally defined function that fills the list box with a list of files in the current directory.

When the dialog function receives the WM_COMMAND message, it looks for three different values: ID_LISTBOX, IDOK, and IDCANCEL.

For ID_LISTBOX, the dialog function checks the notification-message type. If it is LBN_SELCHANGE, the dialog function retrieves the new selection by using the **DlgDirSelect** function. It then copies the new filename to the edit control by using the **SetDlgItemText** function and selects it for editing by sending a EM_SETSEL message. If the current selection is not a filename, the dialog function copies the default specification to the list box by using the **DlgDirList** function. This fills the list box with all files in the current directory.

If the ID_LISTBOX notification type is LBN_DBLCLK, the dialog function carries out the same action as for the IDOK case. A list box sends an LBN_DBLCLK message only after sending an LBN_SELCHANGE message. This means you do not have to retrieve the new filename when you receive a double-click notification.

For IDOK, the dialog function retrieves the contents of the edit control and checks the filename to see if it is valid. The **strchr** function searches for wildcard characters in the name. If it finds a wildcard character, it divides the filename into separate path and filename parts by using the locally defined SeparateFile function. The **strcpy** function updates the DefPath variable with a new default path, if any. The locally defined ChangeDefExt function updates the DefExt variable with a new default filename extension, if any. After the default path, filename, and filename extension are updated, the UpdateListBox function updates the contents of the list box, and the dialog function returns to let the user select a valid filename from the new list.

If a filename has no wildcard characters, the dialog function makes sure the file is not empty. If it is empty, the dialog function displays a warning message, but does not terminate the dialog box. This lets the user try again. If the filename has no wildcards and the file is not empty, and if the user has entered a filename that does not have an extension, the dialog function uses the locally defined AddExt function to append the default filename extension. The dialog function then calls the **EndDialog** function to terminate the modal dialog box and sets the dialog-box return value to NULL.

For IDCANCEL, the dialog function calls the **EndDialog** function to terminate the dialog box and cancel the command. The return value is set to NULL.

The dialog function can also check the existence and access mode of the given file before terminating the dialog box. The existence check, not given in this example, is entirely up to the application. Some simple ways of checking whether a file exists and is accessible are shown in Chapter 11, "File Input and Output."

10.11.6 Add Helper Functions

You need to add several functions to your C-language source file to support the OpenDlg dialog function. These functions are listed as follows:

Function	Description
UpdateListBox	Fills the list box in the Open dialog box with the specified files.
SeparateFile	Divides a pathname into separate path and filename parts.
ChangeDefExt	Copies the filename extension from a filename to a buffer, as long as the extension has no wildcard characters.
AddExt	Appends an extension to a filename if the filename has no filename extension.
_lstrlen	Returns the length of a string.
_lstrcpy	Copies a string to a buffer.
_lstrncpy	Copies a specified number of characters from a string to a buffer.

The UpdateListBox function builds a pathname by using the default path and filename, then passes this pathname to the list box by using the **DlgDirList** function. This function fills the list box with the names of the files and directories identified by the pathname. Add the following statements to the C-language source file:

```
void UpdateListBox(hDlg)
HWND hDlg;
{
    strcpy(str, DefPath);
    strcat(str, DefSpec);
    DlgDirList(hDlg, str, ID_LISTBOX, ID_PATH, 0x4010);
    SetDlgItemText(hDlg, ID_EDIT, DefSpec);
}
```

The **SetDlgItemText** function copies the default filename to the dialog box's edit control.

The SeparateFile function divides a pathname into two parts and copies them to separate buffers. It first moves to the end of the pathname and uses the **AnsiPrev** function to back through it, looking for a drive or directory separator. Add the following statements to your C-language source file:

```
void SeparateFile(hDlg, lpDestPath, lpDestFileName, lpSrcFileName)
HWND hDlg;
LPSTR lpDestPath, lpDestFileName, lpSrcFileName;
```

```
{
    LPSTR lpTmp;

    lpTmp = lpSrcFileName + (long) _lstrlen(lpSrcFileName);

    while (*lpTmp != ':' && *lpTmp != '\' && lpTmp > lpSrcFileName)
        lpTmp = AnsiPrev(lpSrcFileName, lpTmp);

    if (*lpTmp != ':' && *lpTmp != '\') {
        _lstrcpy(lpDestFileName, lpSrcFileName);
        lpDestPath[0] = 0;
        return;
    }
    _lstrcpy(lpDestFileName, lpTmp + 1L);
    _lstrncpy(lpDestPath, lpSrcFileName,
        (int) (lpTmp - lpSrcFileName) + 1);
    lpDestPath[(lpTmp - lpSrcFileName) + 1] = 0;
}
```

The ChangeDefExt, AddExt, _lstrlen, _lstrcpy, and _lstrncpy functions all use standard C-language statements to carry out their tasks. Add the following statements to the C-language source file:

```
void ChangeDefExt(Ext, Name)
PSTR Ext, Name;
{
    PSTR pTptr;

    pTptr = Name;
    while (*pTptr && *pTptr != '.')
        pTptr++;
    if (*pTptr)                         /* true if this is an extension */
        if (!strchr(pTptr, '*') && !strchr(pTptr, '?'))
            strcpy(Ext, pTptr);             /* Copies the extension */
}

void AddExt(Name, Ext)
PSTR Name, Ext;
{
    PSTR pTptr;

    pTptr = Name;
    while (*pTptr && *pTptr != '.')
        pTptr++;
    if (*pTptr != '.')          /* If no extension, add the default */
        strcat(Name, Ext);
}

int _lstrlen(lpStr)
LPSTR lpStr;
{
    int i;
    for (i=0; *lpStr++; i++);      /* Gets length using far pointer */
    return (i);
}

void _lstrncpy(lpDest, lpSrc, n)
LPSTR lpDest, lpSrc;
int n;
```

```
{
    while (n--)
        if(!(*lpDest++ = *lpSrc++))
            return;
}

void _lstrcpy(lpDest, lpSrc)
LPSTR lpDest, lpSrc;
{
    while(*lpDest++ = *lpSrc++);
}
```

10.11.7 Export the Dialog Function

You need to export the OpenDlg dialog function. Add the following line to the **EXPORTS** statement in your module-definition file:

```
OpenDlg @3
```

10.11.8 Compile and Link

No changes are required to the **make** file. Compile and link the application, start Windows, then run the FileOpen application. When you open the File menu and choose the Open command, you will see a dialog box similar to the one shown in Figure 10.1:

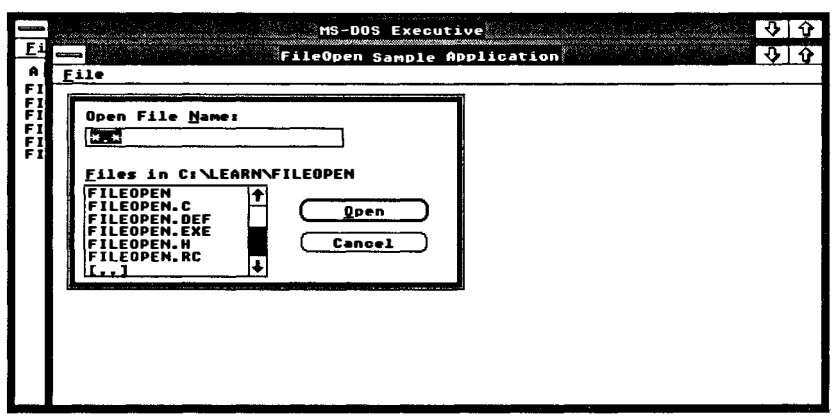


Figure 10.1 FileOpen Dialog Box

Select a file from the list box, or enter a filename in the edit control, then choose the Open button.

Chapter 11
File Input and Output

11.1 Introduction

Although file input and output in Microsoft Windows is similar to file input and output in standard C run-time programs, there are enough differences to make a review of file input and output important. For example, although you can use C run-time, stream input and output (I/O) functions in Windows, the low-level, C run-time input and output functions are preferred. Also, since Windows is multitasking, you need to take special care to manage your open files.

To support these differences, Windows provides the **OpenFile** function. **OpenFile** opens and manages your files, returning a file handle that you can use with the low-level, C run-time functions to read and write data. This chapter explains how to use the **OpenFile** function to open and create disk files. It also explains how to use the low-level, C run-time input and output functions to read from and write to disk files.

11.2 Multitasking and Files

Multitasking imposes some special restrictions on file access that you do not encounter in standard C programs. Since there may be several applications working with files at the same time, you need to follow some simple rules to avoid conflicts and potential overwriting of files.

11.2.1 Open Files

You should keep a file open for only as long as you have execution control. This means you should close the file before calling the **GetMessage** function or any other function that may yield execution control. The reason for closing the file is to prevent it from being affected by changes in the disk environment that may be caused by other applications. For example, if you are writing to a floppy disk and temporarily relinquish control to another application, that application may direct you to remove the floppy disk and replace it with another. If you get control back and attempt to write as before without having closed and reopened the file, you will end up destroying data on the new disk.

Another reason to keep files closed is the DOS open-file limit. DOS sets a limit on the number of open files that can exist at any one time. If many applications attempt to open and use files, they can quickly exhaust the available files.

To prevent open-file problems, the **OpenFile** function provides an OF_REOPEN option that lets you easily close and reopen files. Whenever you open or create a file, **OpenFile** automatically copies the relevant facts about the file, including the full pathname and the current position of the file pointer, in an **OFSTRUCT** structure. This means you can close the file, then reopen it by supplying nothing more than the structure.

If you have changed disks while working in another application, the **OpenFile** function will fail to reopen the file. If you specify the OF_PROMPT option when reopening a file, **OpenFile** automatically displays a message box asking you to insert the correct disk.

11.2.2 Filenames

Ultimately, Windows depends on the DOS file-handling functions to carry out all file input and output. This means that you must follow DOS conventions when carrying out file operations. For example, with DOS, a filename can have from one to eight characters and a filename extension can have from zero to three characters. The name must not contain spaces or special-purpose characters. Furthermore, filenames must be specified in the OEM character set, not the Windows default character set, ANSI.

It is up to you to make sure your filenames are the appropriate length and contain the appropriate characters, but you do not have to worry about translating character sets if you use the **OpenFile** function. For convenience, **OpenFile** automatically translates filenames from the ANSI character set to the OEM set. It does so using the **AnsiToOem** function.

All edit controls and list boxes use the ANSI character set by default, so if you plan to display DOS filenames or let users enter filenames, they may see unexpected characters wherever an OEM character is not identical to an ANSI character.

If you intend to process international filenames, you must be prepared to handle filenames that do not contain conventional single-byte character values. For such filenames, you should use the **AnsiNext** and **AnsiPrev** functions to move forward and backward in a string. These functions correctly handle strings that contain characters that are not one byte in length, such as strings in machines that are using Japanese characters.

11.2.3 Temporary Filenames

Since multiple instances of one application may be running at the same time, one instance can end up overwriting the temporary or scratch file of another instance if you do not use unique filenames for each instance. You can create unique filenames by using the **GetTempFilename** function. This function creates a unique name by combining a unique integer with a prefix and filename extension that you supply. The temporary names follow the DOS filename requirements.

The **GetTempFileName** function uses the TEMP environment variable to create the full pathname of the temporary file. If the user has not set the variable, the temporary file will be placed in the root directory of the current drive. If the variable does not specify a valid directory, you will not be able to create the temporary file.

11.2.4 Errors When Files Are Open

Since an application should not relinquish control while it has open files, applications that need to display an alert or error message by using the **MessageBox** function should either make the message box system-modal, or close the files before displaying the message box. If the message box is not system-modal or the files are not closed, the user can move to another application, taking control away from the application with open files.

11.3 Creating Files

You can use the **OpenFile** function to create a new file. You must supply a null-terminated filename, a buffer having **OFSTRUCT** type, and the OF_CREATE option. The following example creates the *file.txt* file and returns a handle to the file that can be used in low-level, C run-time I/O functions:

```
int hFile;
OFSTRUCT OfStruct;
          .
          .
          .
hFile = OpenFile("file.txt", &OfStruct, OF_CREATE);
```

The **OpenFile** function creates the file, if necessary, and opens it for writing. If the file already exists, the function truncates it to zero length and opens it for writing.

If you wish to prevent overwriting an existing file, you can check whether the file exists, before creating a new file, by calling **OpenFile** as follows:

```
hFile = OpenFile("file.txt", &OfStruct, OF_EXIST);
if (hFile >= 0) {
    wAction = MessageBox(hWnd,
        (LPSTR) "File exists. Overwrite?",
        (LPSTR) "File",
        MB_OKCANCEL);
    if (wAction == IDCANCEL)

/* End this processing */
    }
}

/* Open the file */
```

11.4 Opening Existing Files

You can open an existing file by using the OF_READ, OF_WRITE, or OF_READWRITE options. These options direct the **OpenFile** function to open existing files for reading, writing, or reading and writing. The following example opens the *file.txt* file for reading:

```
hFile = OpenFile("file.txt", &OfStruct, OF_READ);
```

If the file fails to open, you can display a dialog box to indicate that the file was not found. You can also use **OpenFile** to prompt for the file, as described in section 11.7, "Prompting for Files."

11.5 Reading and Writing Files

Once you have opened a file, you can read from it or write to it using low-level, C run-time functions. The following example opens the *file.txt* file for reading and then reads 512 bytes from it:

```
char buffer[512];
int count;
        .
        .
        .
hFile = OpenFile("file.txt", &OfStruct, OF_READ);
if (hFile >= 0) {
    count = read(hFile, buffer, 512);
    close(hFile);
}
```

In this example, the file handle is checked before bytes are read from the file. **OpenFile** returns –1 if the file could not be found or opened. The **close** function closes the file immediately after reading.

The following example opens the *file.tmp* file for writing and then writes bytes from the character-array buffer:

```
hFile = OpenFile("file.tmp", &OfStruct, OF_READ);
if (hFile >= 0) {
    write(hFile, buffer, count);
    close(hFile);
}
```

You should always close floppy-disk files after reading or writing. This is to prevent problems if you remove the current disk while working with another application. You can always reopen a disk file by using the MF_REOPEN option.

11.6 Reopening Files

If you open a file on a floppy disk, you should close it before your application relinquishes control to another application. The most convenient time is immediately after reading or writing the file. The file can always be reopened using **OpenFile** and the OF_REOPEN option:

```
hFile = OpenFile((LPSTR) NULL, &OfStruct, OF_REOPEN | OF_READ);
```

In this example, **OpenFile** uses the filename in the OfStruct structure to open the file. When a file is reopened, the file pointer marking the current position in the file is moved to the same position it was in just before the file was closed.

11.7 Prompting for Files

You can automatically prompt the user to insert the correct disk before reopening a file by using the OF_PROMPT option. **OpenFile** uses the filename to create a prompt string. If you are reopening a file, you need to use the OF_REOPEN and OF_PROMPT options in addition to specifying how you want to open the file:

```
hFile = OpenFile((LPSTR)NULL, &OfStruct, OF_PROMPT | OF_REOPEN
    | OF_READ);
```

If you reopen a file as read only, Windows will check whether the date and time match the date and time of the file when it was first opened.

11.8 Checking File Status

You can retrieve the current file status of a open file by using the low-level, C run-time function, **fstat**. This function fills a structure with information about a file, such as its length in bytes (specified in the **size** field) and the date and time it was created. The following example fills the FileStatus structure with information about the *file.txt* file:

```
stat FileStatus;
        .
        .
        .
fstat(hFile, FileStatus);
```

11.9 A Simple File Editor: EditFile

This example shows how to create a simple Windows application that uses the **OpenFile** and C run-time functions to open and save small text files. To create the EditFile sample application, you will need to copy and rename the FileOpen application sources, described in Chapter 10, "Controls and Dialog Boxes," and modify them as follows:

1. Add constants to the include file.
2. Create a SaveAs dialog-box template and add it to the resource script file.
3. Add new include statements to the C-language source file.
4. Add new variables.
5. Replace the WM_COMMAND case.
6. Add the WM_QUERYENDSESSION and WM_CLOSE cases.
7. Modify the OpenDlg dialog function.
8. Create a SaveAs dialog function.
9. Create helper functions for the SaveAs dialog function.
10. Export the SaveAs dialog function.
11. Modify the application's **HEAPSIZE** statement.
12. Compile and link the application.

When this application is completed, you will be able to view text files in an edit control. The application's Open command in the File menu will let you specify the file to be opened. You will also be able to make changes to a file or enter new text, and save the text using the Save or Save As command in the dialog box.

11.9.1 Add Constants to the Include File

You need to add a constant definition to the include file to support the SaveAs dialog box. Add the following statement to the include file:

```
#define MAXFILESIZE 0x7FFF
```

11.9.2 Add a SaveAs Dialog Box

You need a new dialog box to support the Save As command. The SaveAs dialog box prompts for a filename, and lets the user enter the name in an edit control. Add the following **DIALOG** statement to the resource file:

```
SaveAs DIALOG 10, 10, 180, 53
STYLE WS_DLGFRAME | WS_POPUP
BEGIN
    LTEXT "Save As File &Name:", ID_FILENAME, 4,  4,  72, 10
    LTEXT "",                    ID_PATH,    84,  4,  92, 10
    EDITTEXT                     ID_EDIT,     4, 16, 100, 12
    DEFPUSHBUTTON  "Save",       IDOK,      120, 16,  50, 14
    PUSHBUTTON     "Cancel",     IDCANCEL,  120, 36,  50, 14
END
```

The constants, ID_PATH, ID_FILENAME, ID_EDIT, IDCANCEL, and IDOK, are the same as those used in the Open dialog box. Since the Open and SaveAs dialog boxes will never be open at the same time, there is no need to worry about conflicting control IDs.

11.9.3 Add Include Statements

You need to include additional C run-time include files to support the file input and output operations. Add the following statements to the beginning of the C-language source file:

```
#include <sys\types.h>
#include <sys\stat.h>
```

11.9.4 Add New Variables

To open and save a file you will need to declare some global variables. The following variables should be declared at the beginning of the file:

```
HANDLE hEditBuffer;          /* handle to editing buffer        */
HANDLE hOldBuffer;           /* old buffer handle               */
HANDLE hHourGlass;           /* handle to hourglass cursor      */
HANDLE hSaveCursor;          /* current cursor handle           */
int hFile;                   /* file handle                     */
int count;                   /* number of chars read or written */
PSTR pBuffer;                /* address of read/write buffer    */
OFSTRUCT OfStruct;           /* information from OpenFile()     */
```

```
struct stat FileStatus;          /* information from fstat()        */
BOOL bChanges = FALSE;           /* TRUE if the file is changed     */
BOOL bSaveEnabled = FALSE;       /* TRUE if text in the edit buffer */
PSTR pEditBuffer;                /* address of the edit buffer      */

char Untitled[] =                /* default window title            */
    "Edit File - (untitled)";
```

The hEditBuffer variable holds the handle of the current editing buffer. This buffer, located in the application's heap, contains the current file text. To load a file, you allocate the buffer, load the file, then pass the buffer handle to the edit control. The hOldBuffer variable is used to replace an old buffer with a new one. The hHourGlass and hSaveCursor handles hold cursor handles for lengthy operations.

The hFile variable holds the file handle returned by the **OpenFile** function. The count variable holds a count of the number of characters to be read or written. The pBuffer variable is a pointer, and holds the address of the character that contains the characters to be read or written. The OfStruct structure holds information about the file.

The FileStatus structure holds information about the file. The bChanges variable is TRUE if the user has changed the contents of the file. The bSaveEnabled variable is TRUE if the user has given a valid name for the file to be saved. The Untitled variable holds the main window's caption, which changes whenever a new file is loaded.

11.9.5 Replace the WM_COMMAND Case

You need to replace the WM_COMMAND case to process all File-menu commands except Print. For the New command, you will need to clear the current filename and empty the edit control if there is any text in it. For the Open command, you need to retrieve the selected filename, open the file, and fill the edit control. For the Save command, you need to write the contents of the edit control back to the current file. Finally, for the Save As command, you need to prompt for a filename and write the contents of the edit control.

If the user chooses the New command and there is text in the current file that has been modified, you should prompt the user with a message box to determine whether the changes should be saved. Add the following statements to the WM_COMMAND case:

```
case IDM_NEW:
    if (!QuerySaveFile(hWnd))
        return (NULL);
    bChanges = FALSE;
    FileName[0] = 0;
    SetNewBuffer(hWnd, NULL, Untitled);
    break;
```

The locally defined QuerySaveFile function checks the file for changes and prompts the user to save the changes. If the changes are saved, the filename is cleared and the editing buffer is emptied by using the locally defined function, SetNewBuffer.

If the user chooses the Open command and there is text in the current file that has been modified, you should prompt the user to determine whether the changes should be saved before opening the new file. Add the following statements to the WM_COMMAND case:

```
case IDM_OPEN:
    if (!QuerySaveFile(hWnd))
        return (NULL);
    lpOpenDlg = MakeProcInstance((FARPROC) OpenDlg, hInst);
    hFile = DialogBox(hInst, "Open", hWnd, lpOpenDlg);
    FreeProcInstance(lpOpenDlg);
    if (!hFile)
        return (NULL);
    hEditBuffer =
        LocalAlloc(LMEM_MOVEABLE | LMEM_ZEROINIT,
            FileStatus.st_size+1);
    if (!hEditBuffer) {
        MessageBox(hWnd, "Not enough memory.",
            NULL, MB_OK | MB_ICONQUESTION);
        return (NULL);
    }
    hSaveCursor = SetCursor(hHourGlass);
    pEditBuffer = LocalLock(hEditBuffer);
    IOStatus = read(hFile, pEditBuffer, FileStatus.st_size);
    close(hFile);
    if (IOStatus != FileStatus.st_size) {
        sprintf(str, "Error reading %s.", FileName);
        SetCursor(hSaveCursor);            /* Remove the hourglass */
        MessageBox(hWnd, str, NULL,
            MB_OK | MB_ICONQUESTION);
    }
    LocalUnlock(hEditBuffer);
    sprintf(str, "EditFile - %s", FileName);
    SetNewBuffer(hWnd, hEditBuffer, str);
    SetCursor(hSaveCursor);                  /* Restore the cursor */
    break;
```

When the IDM_OPEN case is processed, the QuerySaveFile function checks the existing file for changes before displaying the Open dialog box. The **DialogBox** function now returns a file handle to the open file. This handle is created in the OpenDlg dialog function. If the file can't be opened, the function returns NULL and processing ends. Otherwise, the **LocalAlloc** function allocates the space needed to load the file into memory. The amount of space needed is determined by the FileStatus structure, which is filled with information about the open file by the OpenDlg dialog function. If there is no available memory, a message box is displayed and processing ends. Otherwise, the **SetCursor** function displays the hourglass, the **LocalLock** function locks the new buffer, and the C run-time **read** function copies the contents of the file into memory. If the file was not read completely, a message box is displayed. **SetCursor** restores the cursor before the **MessageBox** function is called. The

LocalUnlock function unlocks the editing buffer, and after a new window caption is created, the locally defined SetNewBuffer function changes the editing buffer and caption.

If the user chooses the Save command and there is no current filename, you should carry out the same action as the Save As command. Add the following statements to the WM_COMMAND case:

```
case IDM_SAVE:
    if (!FileName[0])
        goto saveas;
    if (bChanges)
        SaveFile(hWnd);
    break;
```

The IDM_SAVE case checks for a filename and, if none exists, skips to the IDM_SAVEAS case. If a filename does exist, the locally defined SaveFile function saves the file only if changes have been made to it.

The Save As command should always prompt for a filename. You should save the file only if the user gives a valid filename.

```
case IDM_SAVEAS:
saveas:
    lpSaveAsDlg = MakeProcInstance(SaveAsDlg, hInst);
    Success = DialogBox(hInst, "SaveAs", hWnd, lpSaveAsDlg);
    FreeProcInstance(lpSaveAsDlg);
    if (Success == IDOK) {
        sprintf(str, "EditFile - %s", FileName);
        SetWindowText(hWnd, str);
        SaveFile(hWnd);
    }
    break;                          /* User canceled */
```

The **DialogBox** function displays the SaveAs dialog box. The **MakeProcInstance** and **FreeProcInstance** functions create and free the procedure-instance address for the SaveAsDlg dialog function. The **DialogBox** function returns IDOK from the SaveAsDlg dialog function if the user enters a valid filename. The **SetWindowText** function then changes the window caption, and the SaveFile function saves the contents of the editing buffer to the file.

The Exit command should now prompt the user to determine whether the current file should be saved. Also, to keep track of the changes to the file, you should process notification messages from the edit-control window. Modify the IDM_EXIT case and add the ID_EDIT case to the WM_COMMAND case, as follows:

```
case IDM_EXIT:
    QuerySaveFile(hWnd);
    DestroyWindow(hWnd);
    break;
```

```
case ID_EDIT:
    if (HIWORD(lParam) == EN_CHANGE)
        bChanges = TRUE;
    return (NULL);
```

11.9.6 Add the WM_QUERYENDSESSION and WM_CLOSE Cases

You need to process the WM_QUERYENDSESSION and WM_CLOSE messages to prevent the contents of your files from being lost. Add the following statements to the window function:

```
case WM_QUERYENDSESSION:                /* message: to end the session? */
    return (QuerySaveFile(hWnd));

case WM_CLOSE:                          /* message: close the window    */
    if (QuerySaveFile(hWnd))
        DestroyWindow(hWnd);
    break;
```

Windows sends a WM_QUERYENDSESSION message to the window function when the user has chosen the End Session, Exit, or Close command in the MS-DOS Executive. The session ends only if TRUE is returned. The QuerySaveFile function checks for changes to the file, saves them if desired, and returns TRUE or FALSE depending on whether the user canceled or confirmed the operation.

Windows sends the WM_CLOSE message to the window function when the user has chosen the Close command in the main window's system menu. The QuerySaveFile function carries out the same task as in the WM_QUERYENDSESSION message, but in order to complete the WM_CLOSE case, you must also destroy the main window by using the **DestroyWindow** function.

11.9.7 Modify the OpenDlg Dialog Function

You need to modify the IDOK case in the OpenDlg function in order to open and check the size of the file that is selected by the user. Add the following statements immediately after the call to the AddExt function in the IDOK case of the OpenDlg function:

```
if ((hFile = OpenFile(OpenName, (LPOFSTRUCT) &OfStruct,
        OF_READ)) < 0) {
    sprintf(str, "Error %d opening %s.",
        OfStruct.nErrCode, OpenName);
    MessageBox(hDlg, str, NULL, MB_OK | MB_ICONQUESTION);
}
else {
    fstat(hFile, &FileStatus);
    if (FileStatus.st_size > MAXFILESIZE) {
        sprintf(str,
```

```
            "Not enough memory to load %s.\n%s exceeds %ld bytes.",
            OpenName, OpenName, MAXFILESIZE);
        MessageBox(hDlg, str, NULL,
            MB_OK | MB_ICONQUESTION);
        return (TRUE);
    }
    strcpy(FileName, OpenName);
    EndDialog(hDlg, hFile);
    return (TRUE);
```

The **OpenFile** function opens the specified file for reading and, if successful, returns a file handle. If the file cannot be opened, the case displays a message box containing the error number generated by DOS. If the file is opened, the C run-time **fstat** function copies information about the file into the FileStatus structure. The file size is checked to make sure the file does not exceed the maximum size given by the MAXFILESIZE constant. The case displays an error message if the size is too big. Otherwise, the **strcpy** function copies the new name to the FileName variable and the **EndDialog** function terminates the dialog box and returns the file handle, hFile, to the **DialogBox** function.

11.9.8 Add the SaveAsDlg Dialog Function

You need to supply a dialog function for the SaveAs dialog box. The function will retrieve a filename from the edit control and copy the name to the global variable, FileName. The dialog function should look like this:

```
int FAR PASCAL SaveAsDlg(hDlg, message, wParam, lParam)
HWND hDlg;
unsigned message;
WORD wParam;
LONG lParam;
{
    char TempName[128];

    switch (message) {
        case WM_INITDIALOG:
            if (!FileName[0])
                bSaveEnabled = FALSE;
            else {
                bSaveEnabled = TRUE;
                DlgDirList(hDlg, DefPath, NULL, ID_PATH, 0x4010);
                SetDlgItemText(hDlg, ID_EDIT, FileName);
                SendDlgItemMessage(hDlg, ID_EDIT, EM_SETSEL, 0,
                    MAKELONG(0, 0x7fff));
            }
            EnableWindow(GetDlgItem(hDlg, IDOK), bSaveEnabled);
            SetFocus(GetDlgItem(hDlg, ID_EDIT));
            return (FALSE);         /* FALSE since Focus was changed */

        case WM_COMMAND:
            switch (wParam) {
                case ID_EDIT:
                    if (HIWORD(lParam) == EN_CHANGE && !bSaveEnabled)
                    EnableWindow(GetDlgItem(hDlg, IDOK),
                        bSaveEnabled = TRUE);
```

```
                    return (TRUE);
                case IDOK:
                    GetDlgItemText(hDlg, ID_EDIT, TempName, 128);
                    if (CheckFileName(hDlg, FileName, TempName)) {
                        SeparateFile(hDlg, (LPSTR) str, (LPSTR) DefSpec,
                            (LPSTR) FileName);
                        if (str[0])
                            strcpy(DefPath, str);
                        EndDialog(hDlg, IDOK);
                    }
                    return (TRUE);
                case IDCANCEL:
                    EndDialog(hDlg, IDCANCEL);
                    return (TRUE);
            }
            break;
    }
    return (FALSE);
}
```

The WM_INITDIALOG case enables or disables the Save button. The button should be disabled if there is no current filename. The **EnableWindow** function, along with the bSaveEnabled variable, enables or disables the button. If there is a current filename, it should be the proposed name. The **SetDlgItemText** function copies the filename to the edit control, and the **SendDlgItemMessage** function selects the entire name for editing. The **DlgDirList** function sets the ID_PATH control to the current directory. Since there is no list box to fill, no list box ID is given.

The WM_COMMAND case processes notification messages from the controls in the dialog box. When the function receives the EN_CHANGE notification from the edit control, ID_EDIT, it uses the **EnableWindow** function to enable the Save button, if it is not already enabled.

When the function receives a notification from the Save button, it uses the **GetDlgItemText** function to retrieve the filename in the edit control, then checks the validity of the filename by using the locally defined CheckFileName function. This function checks the filename to make sure it contains no path separators or wildcard characters. It then checks to see if the file already exists; if it does, CheckFileName uses the **MessageBox** function to ask the user whether the file should be overwritten. Finally, the dialog function uses the SeparateFile function to copy the filename to the DefSpec and DefPath variables.

11.9.9 Add Helper Functions

You need to add several functions to your C-language source file to support the EditFile application. These functions are as follows:

Function	Description
CheckFileName	Checks a filename for wildcards, adds the default filename extension if one is needed, and checks for the existence of the file.
SaveFile	Saves the contents of the editing buffer in a file.
QuerySaveFile	Prompts the user to save changes if the file has changed without having been saved.
SetNewBuffer	Frees the existing editing buffer and replaces it with a new one.

The CheckFileName function verifies that a filename is not empty and that it contains no wildcards. It also checks to see whether the file already exists by using the **OpenFile** function and the OF_EXIST option. If the file exists, CheckFileName prompts the user to see whether the file should be overwritten. Add the following statements:

```
BOOL CheckFileName(hWnd, pDest, pSrc)
HWND hWnd;
PSTR pDest, pSrc;
{
    PSTR pTmp;

    if (!pSrc[0])
        return (FALSE);      /* Indicates no filename was specified */

    pTmp = pSrc;
    while (*pTmp) {          /* Searches the string for wildcards */
        switch (*pTmp++) {
            case '*':
            case '?':
                MessageBox(hWnd, "Wildcards not allowed.",
                    NULL, MB_OK | MB_ICONQUESTION);
                return (FALSE);
        }
    }

    AddExt(pSrc, DefExt);  /* Adds the default extension if needed */

    if (OpenFile(pSrc, (LPOFSTRUCT) &OfStruct, OF_EXIST) >= 0) {
        sprintf(str, "Replace existing %s?", pSrc);
        if (MessageBox(hWnd, str, "EditFile",
            MB_OKCANCEL | MB_ICONQUESTION) == IDCANCEL);
            return (FALSE);
    }
    strcpy(pDest, pSrc);
    return (TRUE);
}
```

The SaveFile function uses the OF_CREATE option of the **OpenFile** function in order to open a file for writing. The OF_CREATE option directs **OpenFile** to delete the existing contents of the file. The SaveFile function then retrieves a file-buffer handle from the edit control, locks the buffer, and copies the contents to the file. Add the following statements:

```
BOOL SaveFile(hWnd)
HWND hWnd;
{
    BOOL bSuccess;
    int IOStatus;                          /* result of a file write */

    if ((hFile = OpenFile(FileName, &OfStruct,
        OF_PROMPT | OF_CANCEL | OF_CREATE)) < 0) {
        sprintf(str, "Cannot write to %s.", FileName);
        MessageBox(hWnd, str, NULL, MB_OK | MB_ICONQUESTION);
        return (FALSE);
    }
    hEditBuffer = SendMessage(hEditWnd, EM_GETHANDLE, 0, 0L);
    pEditBuffer = LocalLock(hEditBuffer);
    hSaveCursor = SetCursor(hHourGlass);
    IOStatus = write(hFile, pEditBuffer, strlen(pEditBuffer));
    close(hFile);
    SetCursor(hSaveCursor);
    if (IOStatus != strlen(pEditBuffer)) {
        sprintf(str, "Error writing to %s.", FileName);
        MessageBox(hWnd, str, NULL, MB_OK | MB_ICONQUESTION);
        bSuccess = FALSE;
    }
    else {
        bSuccess = TRUE;       /* Indicates the file was saved      */
        bChanges = FALSE;      /* Indicates changes have been saved */
    }
    LocalUnlock(hEditBuffer);
    return (bSuccess);
}
```

The EM_GETHANDLE message, sent by using the **SendMessage** function, directs the edit control to return the handle of its editing buffer. This buffer is located in local memory, so it is locked by using the **LocalLock** function. Once locked, the contents are written to the file by using the C run-time **write** function. The **SetCursor** function displays the hourglass cursor to indicate a lengthy operation. If **write** fails to write all bytes, the SaveFile function displays a message box. The **LocalUnlock** function unlocks the editing buffer before the SaveFile function returns.

The QuerySaveFile function checks for changes to the file and prompts the user to save or delete the changes, or cancel the operation. If the user wants to save the changes, the function prompts the user for a filename by using the SaveAs dialog box. Add the following statements:

```
BOOL QuerySaveFile(hWnd)
HWND hWnd;
{
    int Response;
    FARPROC lpSaveAsDlg;
```

```
    if (bChanges) {
        sprintf(str, "Save current changes: %s", FileName);
        Response = MessageBox(hWnd, str,
            "EditFile",  MB_YESNOCANCEL | MB_ICONQUESTION);
        if (Response == IDYES) {
check_name:
            if (!FileName[0]) {
                lpSaveAsDlg = MakeProcInstance(SaveAsDlg, hInst);
                Response = DialogBox(hInst, "SaveAs",
                    hWnd, lpSaveAsDlg);
                FreeProcInstance(lpSaveAsDlg);
                if (Response == IDOK)
                    goto check_name;
                else
                    return (FALSE);
            }
            SaveFile(hWnd);
        }
        else if (Response == IDCANCEL)
            return (FALSE);
    }
    else
        return (TRUE);
}
```

The SetNewBuffer function retrieves and frees the editing buffer before allocating and setting a new editing buffer. It then updates the edit control window. Add the following statements to the C-language source file:

```
void SetNewBuffer(hWnd, hNewBuffer, Title)
HWND hWnd;
HANDLE hNewBuffer;
PSTR Title;
{
    HANDLE hOldBuffer;

    hOldBuffer = SendMessage(hEditWnd, EM_GETHANDLE, 0, 0L);
    LocalFree(hOldBuffer);
    if (!hNewBuffer)          /* Allocates a buffer if none exists */
        hNewBuffer = LocalAlloc(LMEM_MOVEABLE | LMEM_ZEROINIT, 1);

    SendMessage(hEditWnd, EM_SETHANDLE, hNewBuffer, 0L);
    InvalidateRect(hEditWnd, NULL, TRUE);    /* Updates the buffer */
    UpdateWindow(hEditWnd);
    SetWindowText(hWnd, Title);
    SetFocus(hEditWnd);
    bChanges = FALSE;
}
```

The new text will not be displayed until the edit control repaints its client area. The **InvalidateRect** function invalidates part of the edit control's client area. The NULL argument means that the entire control needs repainting, and TRUE specifies that the background should be erased before repainting. All of this prepares the control for painting. The **UpdateWindow** function causes Windows to send the edit control a WM_PAINT message immediately.

11.9.10 Export the SaveAsDlg Dialog Function

You need to export the SaveAsDlg dialog function. Add the following line to the **EXPORTS** statement in your module-definition file:

```
SaveAsDlg @4
```

11.9.11 Add Space to the Heap

You need to add extra space to the local heap. This space is required to support the edit control, which uses memory from the local heap to store its current text. Make the following change to the module-definition file:

```
HEAPSIZE OxAFFF
```

This statement makes the maximum possible edit-control buffer slightly less than 32,707 (32K–1) bytes. Files larger than this cannot be opened.

11.9.12 Compile and Link

No changes are required to the **make** file. Compile and link the application, start Windows and then the EditFile application. Now, choose the Open command, select a file, and EditFile will read and display the file. If the file is larger than can fit in the window, you can use the DIRECTION keys to scroll left and right or up and down. Figure 11.1 shows a file in the EditFile application:

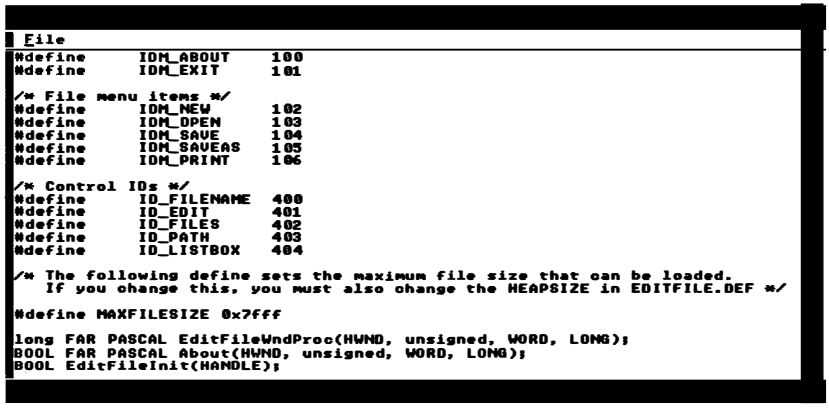


Figure 11.1 EditFile Window

Chapter 12
Printing

12.1 Introduction

In Microsoft Windows, sending output to a printer is similar to sending output to a screen. The only differences are the use of the spooler and the escape sequences.

Printing in Windows is handled by GDI. When an application makes a print request, GDI typically activates the print spooler to queue the print request and make it distinct from other print jobs. Since more than one application can be printing at the same time, the spooler is needed to prevent more than one print job from going to the printer at the same time. Although the spooler is not required in order to print, it is recommended. An application can determine whether GDI will activate the spooler by checking the spooler= line in the [windows] section of *win.ini*, the Windows initialization file.

Escape sequences are required for your application to communicate with the device driver associated with your printer. The **Escape** function tells the device driver what to do, and also gathers information, such as page size, for the application.

When sending output to the printer, you should follow the same general rules as for other types of GDI output. If you are sending text, or primitives, such as rectangles, arcs, and circles, you can send them directly to the printer device context. You can also send text and primitives to a memory device context, before sending them to the printer. This allows you to create complex images in a memory device context before sending them to the printer.

12.2 Using a Printer

This section describes how to use the **Escape** function and other GDI functions to send data from an application to a printer.

12.2.1 Printing a Line of Text

Printing a single line of text requires the following steps:

- Creating the device context for the printer.
- Starting the print request.
- Printing the line.

- Starting a new page.
- Ending the print request.
- Deleting the device context.

The following example shows how to print a single line of text on an Epson FX-80 printer that is connected to the printer port, *lpt1*:

```
hPr = CreateDC("EPSON", "EPSON FX-80", "LPT1:", (LPSTR) NULL);

if (hPr != NULL) {
    Escape(hPr, STARTDOC, 4, (LPSTR) "Test", 0L);
    TextOut(hPr, 10, 10, "A single line of text.", 22);
    Escape(hPr, NEWFRAME, 0, 0L, 0L);
    Escape(hPr, ENDDOC, 0, 0L, 0L);
    DeleteDC(hPr);
}
```

The **CreateDC** function creates the device context for the printer. The following names are required: the name of the device driver, "EPSON"; the name of the device, "EPSON FX-80"; and the name of the printer port, "LPT1:". The last parameter specifies how the printer should be initialized; NULL specifies the default initialization.

The STARTDOC escape, used with the **Escape** function, starts the print request. The name "Test" is used by the spooler to identify the request. Other parameters are not used, so they are set to zero. After the request is started, **TextOut** copies the line of text to the printer. The line will be placed starting at the coordinates (10,10) on the printer paper (the printer coordinates are always relative to the upper-left corner of the paper). The default units are printer pixels. The NEWFRAME escape completes the page and signals the printer to advance to the next page. The ENDDOC escape signals the end of the print request, and the **DeleteDC** function deletes the device context for the printer. In this example, none of the parameters of the NEWFRAME and ENDDOC escapes are required, so they are set to zero.

You should not expect the line of text to be printed immediately. The spooler collects all output for a print request before sending it to the printer, so actual printing does not begin until after the ENDDOC escape.

12.2.2 Printing a Bitmap

You can print a bitmap by following the steps required for printing a single line of text:

1. Create a memory device context that is compatible with the bitmap.

2. Load the bitmap and select it into the memory device context.
3. Start the print request and use the **BitBlt** function to copy the bitmap from the memory device context to the printer.
4. End the print request.
5. Remove the bitmap from the memory device-context selection and delete the device context.

The following example shows how to print a bitmap named "dog" that has been added to the resource file:

```
HDC hDC;
HDC hMemoryDC;
HDC hPr;
BITMAP Bitmap;
          .
          .
          .

hDC = GetDC(hWnd);
hMemoryDC = CreateCompatibleDC(hDC);
ReleaseDC(hWnd, hDC);

hBitmap = LoadBitmap(hInstance, "dog");
GetObject(hBitmap, (LPBITMAP) &Bitmap);
hOldBitmap = SelectObject(hMemoryDC, hBitmap);

hPr = CreateDC("EPSON", "EPSON FX-80", "LPT1:", (LPSTR) NULL);

if (hPr != NULL) {
    Escape (hPr, STARTDOC, 4, (LPSTR) "Dog", OL);
    BitBlt(hPr, 10, 30,
        Bitmap.bmWidth,
        Bitmap.bmHeight,
        hMemDC, 0, 0, SRCCOPY);
    Escape(hPr, NEWFRAME, 0, OL, OL);
    Escape(hPr, ENDDOC, 0, OL, OL);
    DeleteDC(hPr);
}

SelectObject(hMemoryDC, hOldBitmap);
DeleteDC(hMemoryDC);
DeleteObject(hBitmap);
```

In this example, the **CreateCompatibleDC** function creates a memory device context that is compatible with the display context of the current window. The **GetDC** and **ReleaseDC** functions retrieve and release the display context. The **LoadBitmap** function loads the bitmap from the resource file. The **GetObject** function retrieves information about the bitmap, such as its height and width. These values are used later in the **BitBlt** function. The **SelectObject** function selects the bitmap into the memory device context.

The statements for the print request are identical to those in the line-of-text print request, except that the **TextOut** function has been replaced by the **BitBlt** function. **BitBlt** copies the bitmap from the memory device context to the printer, placing the bitmap at the coordinates (10,30).

After the print request is complete, the **SelectObject** and **DeleteDC** functions are used to remove the bitmap from selection and delete the memory device context. Since the bitmap is no longer needed, the **DeleteObject** function removes it from memory.

12.2.3 Retrieving the Current Printer

You can retrieve information about the current printer, such as its type and the computer port it is connected to, by using the **GetProfileString** function. The Windows Control Panel application adds information about the current printer to the "device" field in the [windows] section of the *win.ini* file. Any application can retrieve this information by using the **GetProfileString** function. The information can then be used in the **CreateDC** function to create a printer device context for a particular printer on a particular computer port.

Printer information from the *win.ini* file consists of three fields: the printer type, the printer device-driver name, and the computer port. The fields are separated by commas and, sometimes, spaces. The following example shows how to retrieve the printer information and divide the fields into separate strings:

```
char pPrintInfo[80];
PSTR pTemp;
PSTR pPrintType;
PSTR pPrintDriver;
PSTR pPrintPort;
    .
    .
    .
GetProfileString("windows", "device", pPrintInfo, (LPSTR) NULL, 80);
pTemp = pPrintType = pPrintInfo;
pPrintDriver = pPrintPort = 0;
while (*pTemp) {
    if (*pTemp == ',') {
        *pTemp++ = 0;
        while (*pTemp == ' ')
            pTemp++;
        if (!pPrintDriver)
            pPrintDriver = pTemp;
        else {
            pPrintPort = pTemp;
            break;
        }
    }
    else
        pTemp++;
}
```

```
hPr = CreateDC(pPrintDriver, pPrinterType, pPrintPort, (LPSTR) NULL);
      .
      .
      .
}
```

In this example, the **GetProfileString** function retrieves the device= field from the [windows] section of the *win.ini* file. The function then copies the line to the pPrintInfo array. A **while** statement is used to divide the line into three separate fields: the printer type, the printer device-driver name, and the printer port. The fields are separated by commas, so an **if** statement is used to check for a comma and to replace it with a zero in order to terminate the field. Another **while** statement skips any leading spaces in the next field.

Each pointer—pPrintType, pPrintDriver, and pPrintPort—receives the address of the beginning of its respective field. The pointers are then used in the **CreateDC** function to create a printer device context for the currently selected printer.

12.2.4 Setting Up the Printer

Applications can prepare a printer driver for operation with a particular printer and port by using the **DeviceMode** function for the driver. This function displays a dialog box, letting the user select the printing modes, such as page orientation and paper size, for the printer.

The **DeviceMode** function is actually part of the printer's device driver, and not part of GDI. Therefore, to call the function, you need to load the device driver and retrieve the address of the function by using the **GetProcAddress** function. The application can then use the address to set up the printer.

The following example illustrates how to access the **DeviceMode** function for the Epson FX-80 printer:

```
HANDLE hDriver;
FARPROC lpDeviceMode;
          .
          .
          .

hDriver = LoadLibrary("EPSON.EXE");
lpDeviceMode = GetProcAddress(hDriver, "DEVICEMODE");

if (lpDeviceMode != NULL) {
    (*lpDeviceMode)((HWND)hWnd,     /* handle to parent window */
          (HANDLE)hDriver,          /* handle to driver module */
          (LPSTR)"EPSON FX-80",     /* printer name            */
          (LPSTR)"LPT1:");          /* port name               */
}

FreeLibrary(hDriver);
```

The parent-window handle, hWnd, identifies the application's main window. The driver uses the module handle, hDriver, as its instance handle. The "EPSON FX-80" printer name is required if the driver can access more than one printer model. The "LPT1:" port name specifies the computer port to which the printer is connected. If no printer is connected, the port name should be set to the same name as the printer.

Since each printer has its own modes, each printer driver has its own dialog box that specifies the modes the user can set. The **DeviceMode** function for an Epson FX-80 printer displays a dialog box similar to the one shown in Figure 12.1:

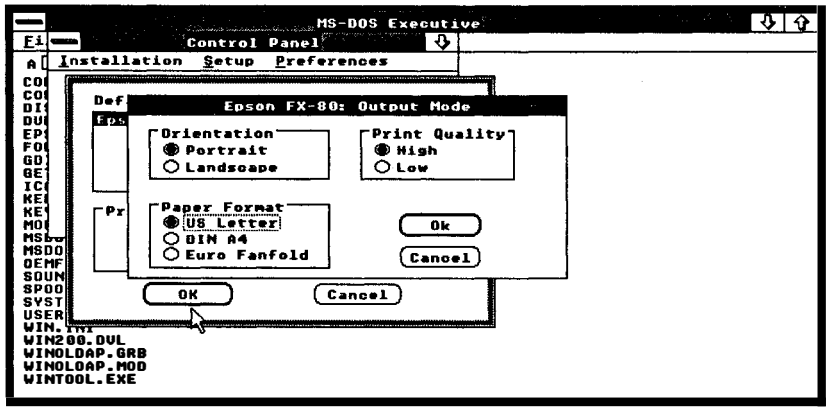


Figure 12.1 Device Modes for an Epson FX-80

The user can set the modes as desired. The driver automatically saves the new modes and prepares the printer.

It is up to the printer driver to determine how printing-mode information should be saved. If the format of the mode information is known, an application can retrieve it by using the **GetEnvironment** function, or set it by using the **SetEnvironment** function or by passing a long pointer to the environment as the last parameter in a **CreateDC** function.

12.2.5 Processing Errors During Printing

Although GDI and the spooler attempt to report all printing errors to the user, applications must be prepared to report out-of-disk and out-of-memory conditions. When there is an error in processing a particular escape, such as STARTDOC or NEWFRAME, the **Escape** function returns a value less than zero. Out-of-disk and out-of-memory errors

usually occur on a NEWFRAME escape. In this case, the return value includes an SP_NOTREPORTED bit. If the bit is clear, GDI has already notified the user. If the bit is set, the application needs to notify the user. The bit is typically set for general-failure, out-of-disk-space, and out-of-memory errors.

The following example shows how you should process unreported errors during printing:

```
status = Escape(hPrDC, NEWFRAME, 0, 0L, 0L);

if (status < 0) {                          /* Any unreported errors? */
        if (status & SP_NOTREPORTED) {                          /* Yes */
            switch (status) {
                case SP_OUTOFDISK:
                    OutOfDiskAlert();
                    break;
                case SP_OUTOFMEMORY:
                    OutOfMemoryAlert();
                    break;
                default:
                    GeneralFailureAlert();
                    break;
            }
        }
        else                /* Reported, but may need further action */
            switch (status) {
                case SP_OUTOFDISK:
                    NoteOutOfDisk();
                    break;
                case SP_OUTOFMEMORY:
                    NoteOutOfMemory();
                    break;
            }
}
```

In this example, if the return value, status, is less than zero and the SP_NOTREPORTED bit in status is set, then unreported error processing is carried out. If status is less than zero but SP_NOTREPORTED is not set, then any reported error processing is carried out.

In the previous example, the correct reponse to an unreported error is to display a message box explaining the error and to terminate the print request. If the error has already been reported, you can terminate the request, then restart it after additional disk or memory space has been made available.

12.2.6 How to Cancel a Print Operation

Applications should always give the user a chance to cancel a lengthy printing operation. To do this, the application needs to create a modeless Abort dialog box when it begins a printing operation. The application also needs to define an abort function that processes messages for the application while the printing operation is in effect.

To create the dialog box and define the abort function, follow these steps:

1. Place the names of the abort function and the dialog function for the Abort dialog box under the **EXPORTS** line of your application's module-definition file:

```
EXPORTS
   AbortDlg      @7
   AbortProc     @8
```

2. Add an Abort-dialog-box template definition to your application's resource script file:

```
AbortDlg DIALOG 20,20,90, 64
STYLE WS_POPUP | WS_DLGFRAME | WS_VISIBLE | WS_CAPTION
CAPTION "PrntFile"
BEGIN
   DefPushButton "Cancel"          IDCANCEL,   29, 44, 32, 14, WS_GROUP
   Ctext    "Sending",             -1,          0,  8, 90,  8
   Ctext    "text",                ID_FILENAME, 0, 18, 90,  8
   Ctext    "to print spooler.",   -1,          0, 28, 90,  8
END
```

3. Use the **MakeProcInstance** function to create a procedure-instance address for each instance of the application:

```
FARPROC lpAbortDlg;
FARPROC lpAbortProc;

lpAbortDlg = MakeProcInstance(AbortDlg, hInstance);
lpAbortProc = MakeProcInstance(fnAbortProc, hInstance);
```

4. Use the SETABORTPROC value to define the abort function to be used during the print operation.
5. Use the **CreateDialog** function to display the Abort dialog box.
6. Use the **EnableWindow** function to disable your parent window.
7. Start the normal print operation, but check the return value from the **Escape** function after each NEWFRAME call. If the value is less than zero, the user has canceled the operation or an error has occurred.
8. Use the **DestroyWindow** function to destroy the Abort dialog box, if necessary. The box destroys itself if the user cancels the print operation.

The following example illustrates how to carry out a printing operation that the user can cancel:

```
HWND hAbortDlgWnd;
BOOL bAbort;
          .
          .
          .

/* Create the printer display context */
```

```
hPr = CreateDC( /* printer parameters */ );

/* Clear the abort flag */
bAbort = FALSE;

/* Create the Abort dialog box (modeless) */
hAbortDlg = CreateDialog(hInstance, (LPSTR) "AbortDlg",
                         hWnd, lpAbortDlg);

/* Disable the main window to avoid reentrancy problems */
EnableWindow(hWnd, FALSE);

/* Define the abort function */
Escape(hPr, SETABORTPROC, 0, lpAbortProc, 0L);

while (/* printing */) {

/* Print one page at a time */
          .
          .
          .
/* Check NEWFRAME for error or abort */
    status = Escape(hPr, NEWFRAME, 0, 0L, 0L);
    if (status < 0 && bAbort)
        break;
    else {
        /* Do error processing */
    }
}

/* End the print operation */
Escape(hPr, ENDDOC, 0, 0L, 0L);

/* Destroy the Abort dialog box */
DestroyWindow(hAbortDlg);

EnableWindow(hWnd, TRUE);
DeleteDC(hPr);
}
```

The abort function retrieves messages from the application queue and dispatches them if they are intended for the Abort dialog box. The function continues to loop until the WM_DESTROY message (generated by the **DestroyWindow** function) is encountered or the print operation is complete. The following example shows the required statements for the abort function:

```
int FAR PASCAL AbortProc(hPr, Code)
HDC hPr;            /* for multiple printer display contexts */
int Code;           /* for printing status */
{
    MSG msg;

/* Process messages intended for the abort dialog box */
    while (PeekMessage((LPMSG) &msg, NULL, NULL, NULL, TRUE))
        if (!IsDialogMessage(hAbortDlgWnd, (LPMSG) &msg)) {
            TranslateMessage((LPMSG) &msg);
            DispatchMessage((LPMSG) &msg);
        }
```

```
/* bAbort is TRUE (return is FALSE) if the user has aborted */
    return (!bAbort);
}
```

The dialog function for the Abort dialog box processes the WM_INITDIALOG and WM_COMMAND messages. To let the user choose the Cancel button with the keyboard, the function takes control of the input focus when the dialog box is initialized. It then ignores all messages until a WM_COMMAND message appears. Command input causes the function to destroy the window and set the abort flag to TRUE. The following example shows the required statements for the dialog function:

```
int FAR PASCAL AbortDlg(hWnd, msg, wParam, lParam)
HWND hWnd;
unsigned msg;
WORD wParam;
LONG lParam;
{

/* Watch for Cancel button, RETURN key, ESCAPE key, or SPACE BAR */
    if (msg == WM_COMMAND) {

/* User has aborted operation */
        bAbort = TRUE;

/* Destroy Abort dialog box */
        DestroyWindow(hWnd);
        return (TRUE);
    }
    else if (msg == WM_INITDIALOG) {

/* Need input focus for user input */
        SetFocus(hWnd);
        return (TRUE);
    }
    return (FALSE);
}
```

12.2.7 The Print Abort Function

Applications that make lengthy print requests are required to pass an abort function to GDI to handle unusual situations during printing operations. The most common situation occurs when a printing operation fills the available disk space before the spooler can copy the data to the printer. Since the spooler can continue to print even though disk space is full, GDI calls the abort function to see if the application wants to cancel the print operation or simply wait until disk space is free.

An application sets the abort function by using the **Escape** function.

```
Escape(hDC, SETABORTPROC, 0, lpAbortProc, 0L)
```

GDI will then call the abort function during spooling. The function must have the following form:

```
int FAR PASCAL AbortProc(hPr, Code)
HDC hPr;
int Code;
```

The hPr argument is a handle to the printer device context, and the Code argument specifies the nature of the call. It can be one of the following:

Code Argument	Description
SP_OUTOFDISK	Spooler has run out of disk space while spooling the data file. The printing operation will continue if the application waits for disk space to become free.
0	Spooler operation is continuing without error.

Once the abort function has been called, it can return TRUE to continue the spooler operation immediately, or return FALSE to cancel the printing operation. Most abort functions call the **PeekMessage** function to temporarily yield control, then return TRUE to continue the print operation. Yielding control typically gives the spooler enough time to free some disk space.

If the abort function returns FALSE, the printing operation is canceled and an error value is returned by the application's next call to the **Escape** function.

Important

If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using the **Escape** function with either the ENDDOC or ABORTDOC escape. GDI automatically terminates the operation before returning the error value.

12.2.8 Canceling a Print Operation with the ABORTDOC Escape

You can use the ABORTDOC escape to cancel a print operation, even if you do not have an abort function or Abort dialog box. In applications that do not have an abort function, ABORTDOC can be used to cancel the operation at any time. In applications that do have abort functions, the ABORTDOC escape can be used only before the first NEWFRAME or NEXTBAND escape.

12.2.9 How to Print Using Banding

Banding is a printing technique in which an image is printed by dividing it into several bands (or slices) and sending each band to the printer separately. Banding lets applications print complex graphics images without first creating the complete image in memory. This can reduce the memory requirements for printing and enhance system performance while printing operations are in effect. Banding can be used on any printing device that has banding capability.

To print using banding, follow these steps:

1. Use the **CreateDC** function to retrieve a device context for the printer.
2. Use the **GetDeviceCaps** function to make sure the printer is a banding device:

   ```
   if (GetDeviceCaps(hPrinterDC, RASTERCAPS) & RC_BANDING)
   ```

3. Use the **Escape** function and the NEXTBAND escape to retrieve the coordinates of a band:

   ```
   Escape(hPrinterDC, NEXTBAND, 0, (LPSTR) NULL, (LPRECT) &rcRect);
   ```

   The function sets the rcRect structure to the coordinates of the current band. Coordinates are in device units, and all subsequent GDI calls are clipped to this rectangle.
4. Check the rcRect structure to see if it is an empty rectangle. The empty rectangle marks the end of the banding operation. If it is empty, terminate the banding operation.
5. Use the **DPtoLP** function to translate the rcRect points from device units to logical units.

   ```
   DPtoLP(hPr, (LPRECT) &rcRect, 2);
   ```

6. Use GDI output and other functions to draw within the band. To save time, the application should carry out only those GDI calls that affect the current band. If an application does not wish to save time, GDI will clip all output that does not appear in the band, so no special action is required.
7. Repeat steps 4 through 6.

Once the banding operation is done, use the **DeleteDC** function to remove the printer device context.

The following example shows how to print using banding:

```
hPr = CreateDC("EPSON", "EPSON FX-80", "LPT1:", (LPSTR) NULL);

if (hPr != NULL) {
    if (GetDeviceCaps(hPr, RASTERCAPS) & RC_BANDING) {
```

```
        Escape(hPr, STARTDOC, 4, (LPSTR) "Dog", 0L);
        Escape(hPr, NEXTBAND, 0, 0L, (LPRECT) &rcRect);
        while (!IsRectEmpty(&rcRect)) {
            DPtoLP(hPr, (LPRECT) &rcRect, 2);

            /* Place your output function here. To save time,
             * use rcRect to determine which functions need
             * to be called for this band.
            */

            Escape(hPr, NEXTBAND, 0, 0L, (LPRECT) &rcRect);
        }
        Escape(hPr, NEWFRAME, 0, 0L, 0L);
        Escape(hPr, ENDDOC, 0, 0L, 0L);
    }
    DeleteDC(hPr);
}
```

12.3 A Sample Application: PrntFile

You can add printing capability to the EditFile application described in Chapter 11, "File Input and Output," by copying the current text from the edit control and printing it by using the methods described in this chapter. To add printing capability, copy and rename the EditFile sources to PrntFile, then modify the sources as follows:

1. Add an AbortDlg dialog-box template to the resource script file.
2. Add new variables for printing.
3. Add the IDM_PRINT case to the WM_COMMAND case.
4. Create the AbortDlg dialog function and AbortProc function.
5. Add the GetPrinterDC function.
6. Export the AbortDlg dialog function and AbortProc function.
7. Compile and link the application.

This example shows how to print the contents of the edit control, including the statements required to support the abort function and the dialog function for the Abort dialog box.

12.3.1 Add an AbortDlg Dialog Box

You need a new dialog box to support printing. The AbortDlg dialog box permits the user to cancel a printing operation by choosing the Cancel button. Add the following **DIALOG** statement to the resource file:

```
AbortDlg DIALOG 20,20,90, 64
STYLE WS_POPUP | WS_DLGFRAME | WS_VISIBLE | WS_CAPTION
```

```
CAPTION "PrntFile"
BEGIN
   DefPushButton "Cancel"          IDCANCEL,    29, 44, 32, 14, WS_GROUP
   Ctext     "Sending",            -1,           0,  8, 90,  8
   Ctext     "text",               ID_FILENAME,  0, 18, 90,  8
   Ctext     "to print spooler.",  -1,           0, 28, 90,  8
END
```

12.3.2 Add Variables for Printing

You need to declare new variables to support printing. Add the following declarations to the beginning of your source file:

```
HDC hPr;                    /* handle for printer device context     */
int LineSpace;              /* spacing between lines                 */
int LinesPerPage;           /* lines per page                        */
int CurrentLine;            /* current line                          */
int LineLength;             /* line length                           */
DWORD dwLines;              /* number of lines to print              */
DWORD dwIndex;              /* index into lines to print             */
char pLine[128];            /* buffer to store lines before printing */
TEXTMETRIC TextMetric;      /* information about character size      */
POINT PhysPageSize;         /* information about the page            */
BOOL bAbort;                /* FALSE if user cancels printing        */
HWND hAbortDlgWnd;
FARPROC lpAbortDlg, lpAbortProc;
```

The hPr variable is the handle for the printer device context. It receives the return value from the **CreateDC** function call. The variables LineSpace and LinesPerPage hold the amount of spacing between lines and the number of lines that can be printed per page, respectively. The CurrentLine variable is a counter that keeps track of the current line on the current page. Lines of text are printed one line at a time. The dwLines variable holds the number of lines in the edit control. The TextMetric structure receives information about the font to be used to print the lines. Only the tmHeight and tmExternalLeading fields are used in this example. The PhysPageSize structure receives the physical width and height of the printer paper. The height is used to determine how many lines per page can be printed.

12.3.3 Add the IDM_PRINT Case

To carry out the printing operation, you need to add an IDM_PRINT case to the main window function. Add the following statements:

```
case IDM_PRINT:
    hPr = GetPrinterDC();
    if (!hPr) {
        sprintf(str, "Cannot print %s", FileName);
        MessageBox(hWnd, str, NULL, MB_OK | MB_ICONQUESTION);
        return (NULL);
    }
    lpAbortDlg =  MakeProcInstance(AbortDlg, hInst);
```

```
lpAbortProc = MakeProcInstance(AbortProc, hInst);
Escape(hPr, SETABORTPROC, NULL,
    (LPSTR) (long) lpAbortProc, (LPSTR) NULL);
if (Escape(hPr, STARTDOC, 4, "PrntFile text",
   (LPSTR) NULL) < 0) {
    MessageBox(hWnd, "Unable to start print job",
        NULL, MB_OK | MB_ICONQUESTION);
    FreeProcInstance(AbortDlg);
    FreeProcInstance(AbortProc);
    DeleteDC(hPr);
}

bAbort = FALSE;                                /* Clears the abort flag */
hAbortDlgWnd = CreateDialog(hInst, "AbortDlg", hWnd, lpAbortDlg);
EnableWindow(hWnd, FALSE);
GetTextMetrics(hPr, &TextMetric);
LineSpace = TextMetric.tmHeight + TextMetric.tmExternalLeading;
Escape(hPr, GETPHYSPAGESIZE, NULL, (LPSTR) NULL, (LPSTR) &PhysPageSize);
LinesPerPage = PhysPageSize.y / LineSpace;
dwLines = SendMessage(hEditWnd, EM_GETLINECOUNT, 0, 0L);
CurrentLine = 1;
for (dwIndex = IOStatus = 0; dwIndex < dwLines; dwIndex++) {
    pLine[0] = 128;                             /* Maximum buffer size */
    pLine[1] = 0;
    LineLength = SendMessage(hEditWnd, EM_GETLINE,
        (WORD) dwIndex, (LONG) ((LPSTR) pLine));
    TextOut(hPr, 0, CurrentLine*LineSpace, (LPSTR) pLine, LineLength);
    if (++CurrentLine > LinesPerPage ) {
        Escape(hPr, NEWFRAME, 0, 0L, 0L);
        CurrentLine = 1;
        IOStatus = Escape(hPr, NEWFRAME, 0, 0L, 0L);
        if (IOStatus < 0 || bAbort)
            break;
    }
}

if (IOStatus >= 0 && !bAbort) {
    Escape(hPr, NEWFRAME, 0, 0L, 0L);
    Escape(hPr, ENDDOC, 0, 0L, 0L);
}
EnableWindow(hWnd, TRUE);
DestroyWindow(hAbortDlgWnd);
FreeProcInstance(AbortDlg);
FreeProcInstance(AbortProc);
DeleteDC(hPr);
break;
```

The locally defined GetPrinterDC function checks the *win.ini* file for the current printer and creates a device context for that printer. If there is not a current printer or the device context cannot be created, the function returns NULL and processing ends with a warning. Otherwise, the **MakeProcInstance** function creates procedure instance addresses for the AbortDlg dialog function and the AbortProc function. The SETABORT-PROC escape used with the **Escape** function sets the abort function. The STARTDOC escape starts the printing job and sets the printing title (shown in the Spooler application). If the STARTDOC escape fails, the **FreeProcInstance** function frees the AbortDlg and AbortProc procedure instances and the **DeleteDC** function deletes the device context before processing ends.

The **CreateDialog** function creates the AbortDlg dialog box and the **EnableWindow** function disables the main window. This prevents users from attempting to work in the main window while printing. They can, however, continue to work in some other application.

Since the edit control may contain more than one line, it is important to provide adequate spacing between lines. This keeps one line from overwriting or touching another. The **GetTextMetrics** call retrieves current font information, such as height and external leading, which can be used to compute adequate line spacing. The height is the maximum height of characters in the font. The external leading is the recommended amount of space, in addition to the height, that should be used to separate lines of text in this font. The line spacing, assigned to the LineSpace variable, is the sum of the height and external leading fields, TextMetric.tmHeight and TextMetric.tmExternalLeading.

Since the edit control may contain more lines than can fit on a single page, it is important to determine how many lines can fit on a page and to advance to the next page whenever this line limit is reached. The GETPHYSPAGESIZE escape retrieves the physical dimensions of the page and copies the dimensions to the PhysPageSize structure. PhysPageSize contains both the width and height of the page. The lines per page, assigned to the LinesPerPage variable, is the quotient of the physical height of the page, PhysPageSize.y, and the line spacing, LineSpace.

The **TextOut** function can print only one line at a time, so a **for** statement provides the loop required to print more than one line of text. The EM_GETLINECOUNT message, sent to the edit control by using the **SendMessage** function, retrieves the number of lines to be printed and determines the number of times to loop. On each execution of the loop, the EM_GETLINE message copies the contents of a line from the edit control to the line buffer, pLine. The loop counter, index, is used with the EM_GETLINE message to specify which line to retrieve from the edit control. The EM_GETLINE message also causes **SendMessage** to return the length of the line. The length is assigned to the LineLength variable.

Once a line has been copied from the edit control, it is printed by using the **TextOut** function. The product of the variables CurrentLine and LineSpacing determines the *y*-coordinate of the line on the page. The *x*-coordinate is set to zero. After a line is output, the value of the CurrentLine variable is increased by one. If CurrentLine is greater than LinesPerPage, it is time to advance to the next page. Any text printed beyond the physical bottom of a page is clipped. There is no automatic page advance, so it is important to keep track of the number of lines printed on a page and to use the NEWFRAME escape to advance to the next page when necessary. If there are any errors during printing, the NEWFRAME escape returns an error number and processing ends.

After all lines in the edit control have been printed, the NEWFRAME escape advances the final page and the ENDDOC escape terminates the print request. The **DeleteDC** function deletes the printer device context since it is no longer needed, and the **DestroyWindow** function destroys the AbortDlg dialog box.

12.3.4 Create the AbortDlg and AbortProc Functions

You need to create the AbortDlg and AbortProc functions to support the printing process. The AbortDlg dialog function provides support for the AbortDlg dialog box that appears while the printing is in progress. The dialog box lets the user cancel the printing operation if necessary. The AbortProc function processes messages intended for the AbortDlg dialog box and terminates the printing operation if the user has requested it.

The AbortDlg dialog function sets the input focus and sets the name of the file being printed. It also sets the bAbort variable to TRUE if the user chooses the Cancel button. Add the following statements to the C-language source file:

```
int FAR PASCAL AbortDlg(hDlg, msg, wParam, lParam)
HWND hDlg;
unsigned msg;
WORD wParam;
LONG lParam;
{
    switch(msg) {
        case WM_COMMAND:
            return (bAbort = TRUE);

        case WM_INITDIALOG:
            SetFocus(GetDlgItem(hDlg, IDCANCEL));
            SetDlgItemText(hDlg, ID_FILENAME, FileName);
            return (TRUE);
          }
    return (FALSE);
}
```

The AbortProc function checks for messages in the application queue and dispatches them to the AbortDlg dialog function or to other windows in the application. If one of these messages causes the AbortDlg dialog function to set the bAbort variable to TRUE, the AbortProc function returns this value, directing Windows to stop the printing operation. Add the following statements to the C-language source file:

```
int FAR PASCAL AbortProc(hPr, Code)
HDC hPr;                /* for multiple printer display contexts */
int Code;               /* printing status                       */
```

```
{
    MSG msg;

    while (!bAbort && PeekMessage(&msg, NULL, NULL, NULL, TRUE))
        if (!IsDialogMessage(hAbortDlgWnd, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    return (!bAbort);
}
```

12.3.5 Add the GetPrinterDC Function

You need to add a function to your C-language source file to support the printing operation. The GetPrinterDC function retrieves the "device" field from the [windows] section of the *win.ini* file, divides the entry into its separate components, then creates a printer device context using the device name and printer port given in the entry. Add the following statements to the C-language source file:

```
HANDLE GetPrinterDC()
{
    char pPrintInfo[80];
    LPSTR lpTemp;
    LPSTR lpPrintType;
    LPSTR lpPrintDriver;
    LPSTR lpPrintPort;

    if (!GetProfileString("windows", "device",
            (LPSTR) "", pPrintInfo, 80))
        return (NULL);
    lpTemp = lpPrintType = pPrintInfo;
    lpPrintDriver = lpPrintPort = 0;
    while (*lpTemp) {
        if (*lpTemp == ',') {
            *lpTemp++ = 0;
            while (*lpTemp == ' ')
                lpTemp = AnsiNext(lpTemp);
            if (!lpPrintDriver)
                lpPrintDriver = lpTemp;
            else {
                lpPrintPort = lpTemp;
                break;
            }
        }
        else
            lpTemp = AnsiNext(lpTemp);
    }

    return (CreateDC(lpPrintDriver, lpPrintType, lpPrintPort, (LPSTR) NULL));
}
```

To separate the "device" field into its three components, the **AnsiNext** function advances through the field one character at a time.

12.3.6 Export the AbortDlg and AbortProc Functions

You need to export the AbortDlg dialog function and the AbortProc function. Add the following lines to your module-definition file under the **EXPORTS** statement:

```
AbortDlg        @5  ; Called so user can cancel the print function
AbortProc       @6  ; Processes messages intended for the Abort dialog box
```

12.3.7 Compile and Link

No changes are required to the **make** file. Compile and link the PrntFile application, then start Windows and activate PrntFile; you will see that the Print command has been added to the File menu. You can print by opening a file or by entering text from the keyboard, then choosing the Print command.

Chapter 13
The Clipboard

13.1 What Is the Clipboard?

The clipboard is the data-exchange feature of Microsoft Windows. It is a common area to store data handles through which applications can exchange formatted data. The clipboard holds any number of different data formats and corresponding data handles, all representing the same data, but in as many different formats as an application is willing to supply. For example, a pie chart might be held in the clipboard as both a metafile picture and a bitmap. An application pasting the pie chart would have to decide which representation it wanted. The general rule is that the one with the most information is the most desirable.

This chapter explains how to use the clipboard to do the following:

- Copy text to the clipboard.
- Paste text from the clipboard.
- Copy a bitmap to the clipboard.
- Paste a bitmap from the clipboard.
- Use the clipboard viewer chain.

13.2 Using the Clipboard

To copy data to the clipboard, you must format the selected data by using either a predefined or private format. For most formats, you must allocate global memory and copy the data into it. You then copy the memory handle to the clipboard by using the **SetClipboardData** function.

In Windows applications, copying and pasting are carried out through Edit-menu commands. You can add the Edit menu to an application by following the steps used to create the EditMenu application described in Chapter 8, "Menus."

Windows provides several predefined data formats for use in data interchange. Following is a list of common formats and their contents:

Format	Contents
CF_TEXT	Null-terminated text
CF_METAFILEPICT	Metafile-picture structure
CF_BITMAP	A bitmap

CF_SYLK	SYLK standard data-interchange format
CF_DIF	DIF standard data-interchange format

When you paste data from the clipboard by using the **GetClipboard** function, you specify the format you expect. The clipboard supplies the data only if it has been copied in that format.

13.2.1 Copying Text to the Clipboard

You can copy a short string of text to the clipboard. To do this you will need to do the following:

- Copy the string to global memory.
- Open the clipboard.
- Clear the clipboard.
- Give the global memory handle to the clipboard.
- Close the clipboard.

You copy text to the clipboard in response to the user choosing the Copy command from the Edit menu. To process the menu input and copy the text string to the clipboard, you need to add a WM_COMMAND case to the window function. Add the following statements:

```
case WM_COMMAND:
    switch (wParam) {
    case ID_COPY:
        hData = GlobalAlloc(GMEM_MOVEABLE, (LONG) strlen(string));
        if (hData == 0)
            break;
        lpData = GlobalLock(hData);      /* Lock it down               */
        for (i = 0; string[i] != 0; i++)/* Can't copy with C run-time */
           lpData[i] = string[i];        /* so use this to copy string */
        lpData[i] = 0;                   /* null terminate             */
        GlobalUnlock (hData);
        if (OpenClipboard(hWnd)) {
            EmptyClipboard();
            SetClipboardData(CF_TEXT, hData);
            CloseClipboard();
        }
        hData = 0;
        break;
   }
   break;
```

When copying text, the clipboard requires the text to be in a global memory block. This means you will need to allocate a global block and copy the string to it. In the WM_COMMAND case, the following statements allocate global memory and copy the string:

```
hData = GlobalAlloc(GMEM_MOVEABLE, (LONG) strlen(string));
if (hData == 0)
    break;
lpData = GlobalLock(hData);          /* Lock it down              */
for (i = 0; string[i] != 0; i++)     /* Can't copy with C run-time */
   lpData[i] = string[i];            /* so use this to copy string */
lpData[i] = 0;                       /* null terminate            */
GlobalUnlock(hData);
```

The **GlobalAlloc** function allocates enough memory to hold the string. The GMEM_MOVEABLE flag specifies movable memory. The clipboard can take either fixed or movable memory, but should not be given discardable memory. Movable memory is the most efficient. You should always check a memory handle to ensure it is valid (not NULL) before attempting to lock it.

As you would do with any movable memory, you must lock it to retrieve the memory address. The locally defined _lstrcpy function is used instead of the C run-time **strcpy** function, since **strcpy** cannot handle mixed pointers (string is a short pointer and lpData is a long pointer). The clipboard requires the string to have a terminating null character. Finally, the memory must be unlocked before it can be copied to the clipboard.

Each time you copy the string to the clipboard, this code will allocate another global memory block. The reason for this is that once you have passed a data handle to the clipboard, the clipboard takes ownership of it. This means that you can no longer use the handle other than to view contents, and you must not attempt to free the handle or change its contents.

Copying the global memory handle to the clipboard is simple: open the clipboard, empty it, set the data handle, then close the clipboard. The following statements carry out these steps:

```
if (OpenClipboard(hWnd)) {
    EmptyClipboard();
    SetClipboardData(CF_TEXT, hData);
    CloseClipboard();
}
hData = 0;
```

The **OpenClipboard** function opens the clipboard for the specified window. **OpenClipboard** will fail if another window already has the clipboard open. The **EmptyClipboard** function clears all existing handles in the clipboard and assigns ownership of the clipboard to the window that has it open. An application must empty the clipboard before copying data to it. The **SetClipboardData** function copies the memory handle to the clipboard and identifies the data format, CF_TEXT. The clipboard is then closed by the **CloseClipboard** function. Since the clipboard now owns the global memory identified by hData, it is convenient to set this memory to zero to prevent attempts to free or change the memory.

13.2.2 Pasting Text from the Clipboard

You can paste text from the clipboard into your client area. That is, you can retrieve a text handle from the clipboard and display it in the client area by using the **TextOut** function. To do this you will need to do the following:

- Open the clipboard.
- Retrieve the data handle associated with CF_TEXT.
- Close the clipboard.

You should let the user paste only if there is text in the clipboard. To prevent attempts to paste when no text is present, you can check the clipboard before Windows displays the Edit menu by processing the WM_INITMENU message. If the clipboard is empty, you can disable the Paste command; if text is present, you can enable it. Add the following statements to the window function:

```
WORD wFormat;
        .
        .
        .
case WM_INITMENU:
    if (wParam == GetMenu(hWnd)) {
        wFormat = 0;
        while ((wFormat = EnumClipboardFormats(wFormat)) != 0
            && wFormat != CF_TEXT)
            ;
        if (wFormat == CF_TEXT)
            EnableMenuItem(wParam, ID_PASTE, MF_ENABLED);
        else
            EnableMenuItem(wParam, ID_PASTE, MF_GRAYED);
    }
    break;
```

Since an application has at least two menus, including a system menu, it is important to ensure that the message applies to the Edit menu. You can do this by using the **GetMenu** function.

The **EnumClipboardFormats** function checks for the CF_TEXT format. The function enumerates the contents of the clipboard by returning the format type of each handle. The **while** statement is required, since the function returns only one format at a time. Finally, the **EnableMenuItem** function enables or disables the Paste command based on whether the CF_TEXT format is found.

You can paste from the clipboard when the user chooses the Paste command from the Edit menu. To process the menu input and retrieve the text from the clipboard, you will need to add an ID_PASTE case to the WM_COMMAND case in the window function. Add the following statements immediately after the ID_COPY case:

```
case ID_PASTE:
    if (OpenClipboard(hWnd)) {
        hClipData = GetClipboardData(CF_TEXT);
        CloseClipboard();
        if (hClipData == 0)
            break;
        hDC = GetDC(hWnd);
        lpClipData = GlobalLock(hClipData);
        for (count = 0; lpClipData[count] != 0; count++)
            ;
        TextOut(hDC, 10, 10, lpClipData, count);
        GlobalUnlock(hClipData);
        ReleaseDC(hWnd, hDC);
    }
    break;
```

The **OpenClipboard** function opens the clipboard for the specified window if it is not already open. The **GetClipboardData** function retrieves the data handle for the text, or it retrieves zero if there is no such data. This handle should be checked before being used. The **CloseClipboard** function closes the clipboard, which should always be closed immediately after it has been used. Closing the clipboard lets other applications access it.

The **GetClipboardData** function returns a handle to global memory. The global memory is assumed to contain a null-terminated ANSI string. This means the global memory can be locked by using the **GlobalLock** function, and the contents can be displayed in the client area by using the **TextOut** function. In this example, the locally defined _lstrlen function is used to count the number of characters in the string instead of the C runtime **strlen** function. Since this application is compiled as a small-model application, **strlen** requires a short pointer and cannot use the long pointer, lpClipData.

So that you will be able to see that your application has copied the contents of the clipboard, the **TextOut** function writes to the coordinates (10,10) in your client area. You will need a display context to use **TextOut**, so the **GetDC** function is required, and since you must release a display context immediately after using it, the **ReleaseDC** function is also required.

This method of displaying the text in the client area is for illustration only. Since the content of the string is not saved by the application, there is no way to repaint the text if the client-area background is erased, such as during processing of a WM_PAINT message.

You must not modify or delete the data you have retrieved from the clipboard. You can examine it or make a copy of it, but you must not change it. To examine the data, you may need to lock the handle, as in this example, but you must never leave a data handle locked. Unlock it immediately after using it.

Data handles returned by the **GetClipboardData** function are for temporary use only. Handles belong to the clipboard, not to the application requesting data. Accordingly, handles should not be freed and should be unlocked immediately after using. The application should not rely on the handle's remaining valid indefinitely. In general, the application should copy the data associated with the handle, then release it without changes.

13.2.3 Pasting Bitmaps from the Clipboard

You can paste more than just text into your client area from the clipboard. You can retrieve a bitmap from the clipboard and display it in your client area. To retrieve and display a bitmap, you use the same technique as for pasting text but you make a few changes to accommodate bitmaps.

First, you must modify the WM_INITMENU case in the window function so that it recognizes the CF_BITMAP format instead of CF_TEXT. After you change it, the WM_INITMENU case should look like this:

```
case WM_INITMENU:
    if (wParam == GetMenu(hWnd)) {
        wFormat = 0;
        while ((wFormat = EnumClipboardFormats(wFormat)) != 0
            && wFormat != CF_BITMAP)
            ;
        if (wFormat == CF_BITMAP)
            EnableMenuItem(wParam, ID_PASTE, MF_ENABLED);
        else
            EnableMenuItem(wParam, ID_PASTE, MF_GRAYED);
    }
    break;
```

Retrieving a bitmap from the clipboard is as easy as retrieving text, but displaying a bitmap requires more work than does displaying text. In general, you need to do the following:

1. Retrieve the bitmap data handle from the clipboard. Bitmap data handles from the clipboard are GDI bitmap handles (created by using functions such as **CreateBitmap**).
2. Create a compatible display context and select the data handle into the compatible display context.
3. Use the **BitBlt** function to copy the bitmap to the client area.
4. Release the bitmap handle from the current selection.

After you have changed it, the ID_PASTE case should look like this:

```
case ID_PASTE:
    if (OpenClipboard(hWnd)) {
        hClipData = GetClipboardData(CF_BITMAP);
        CloseClipboard();
        if (hClipData == 0)
```

```
            break;
        hDC = GetDC(hWnd);
        hMemoryDC = CreateCompatibleDC(hDC);
        if (hMemoryDC != NULL) {
            GetObject(hClipData, sizeof(BITMAP),
                (LPSTR) &PasteBitmap);
            hOldBitmap = SelectObject(hMemoryDC, hClipData);
            if (hOldBitmap != NULL)
                BitBlt(hDC, 10, 10,
                    PasteBitmap.bmWidth,
                    PasteBitmap.bmHeight,
                    hMemoryDC, 0, 0, SRCCOPY);
                SelectObject(hMemoryDC, hOldBitmap);
            }
            DeleteDC(hMemoryDC);
        }
        ReleaseDC(hWnd, hDC);
    }
    break;
```

The **CreateCompatibleDC** function returns a handle to a display context, in memory, that is compatible with your computer's display. This means any bitmaps that you select for this display context can be copied directly to the client area. If **CreateCompatibleDC** fails (returns NULL), the bitmap cannot be displayed.

The **GetObject** function retrieves the width and height of the bitmap, as well as a description of the bitmap format. It copies this information into the PasteBitmap structure, whose size is specified by the **sizeof** function. In this example, only the width and height are used and then only in the **BitBlt** function. The **SelectObject** function selects the bitmap into the compatible display context. If it fails (returns NULL), the bitmap cannot be displayed. **SelectObject** may fail if the bitmap has a different format than that of the compatible display context. This can happen, for example, if the bitmap was created for a display on some other computer.

The **DeleteDC** function removes the compatible display context. Before a display context can be deleted, its original bitmap must be restored by using the **SelectObject** function.

13.2.4 Selecting and Copying Bitmaps to the Clipboard

Whether you are copying text or bitmaps to the clipboard, you will want to give the user some method of selecting the data to be copied. The goal is to let the user select a portion of the client area for copying to the clipboard. You can use the same technique described in Chapter 9, "Bitmaps," to select a portion of the client area by using the mouse. The steps required to make the selections are as follows:

1. When the WM_LBUTTONDOWN message is received, use the locally defined StartSelection function to save the current mouse location and begin selection.

2. On each WM_MOUSEMOVE message, use the locally defined UpdateSelection function to make sure the rectangle described by the original WM_LBUTTONDOWN location and the current location is inverted (in reverse video).
3. When the WM_LBUTTONUP message is received, use the locally defined EndSelection function to save the current mouse position. The rectangle described by the original position and this final position marks the area to be copied.

The steps required to copy the bitmap are as follows:

1. Restore the screen (remove the reverse video).
2. Create a compatible display context; create and select a compatible bitmap; and copy the screen bitmap to the compatible display.
3. Release the compatible bitmap from the selection and copy it to the clipboard.

The following statements are necessary to tell the application what to do when a user chooses the Copy command:

```
case ID_COPY:
    hDC = GetDC(hWnd);
    PatBlt(hDC, OrgX, OrgY,
        CopyWidth,
        CopyHeight,
        DSTINVERT);
    hMemoryDC = CreateCompatibleBitmap(hDC);
    if (hMemoryDC != NULL) {
        hBitmap = CreateCompatibleBitmap(hDC,
            CopyWidth, CopyHeight);
        if (hBitmap != NULL) {
            hOldBitmap = SelectObject(hMemDC, hBitmap);
            if (hBitmap != NULL) {
                BitBlt(hMemoryDC, 0, 0, CopyWidth, CopyHeight,
                    hDC, OrgX, OrgY, SRCCOPY);
            hBitmap = SelectObject(hMemDC, hOldBitmap);
            if (hBitmap != NULL && OpenClipboard(hWnd)) {
                EmptyClipboard();
                SetClipboardData(CF_BITMAP, hBitmap);
                CloseClipboard();
                }
            }
        }
        DeleteDC(hMemoryDC);
    }
    ReleaseDC(hWnd, hDC);
    EnableMenuItem(hMenu, ID_COPY, MF_GRAYED);
    OrgX = OrgY = CopyWidth = CopyHeight = 0;
    break;
```

The **GetDC** and **PatBlt** functions clear the selection rectangle from the screen. This is required so that a true bitmap is copied instead of an inverted one.

The **CreateCompatibleDC** and **CreateCompatibleBitmap** functions prepare the compatible display context that will receive the bitmap from the screen. A bitmap cannot be copied directly from the screen to the clipboard. It must be copied to a compatible bitmap first. The handle of the compatible bitmap is then copied to the clipboard.

To copy a bitmap from the screen, the **SelectObject** function selects the compatible bitmap into the compatible display context. The **BitBlt** function then copies the screen image to the compatible display context and, hence, the compatible bitmap. To copy the bitmap to the clipboard, you must remove the bitmap from selection in the compatible display context. You can then copy it to the clipboard by using the same sequence of functions you would use to copy text to the clipboard. In each step of the operation, if the bitmap or display context cannot be created, the copy operation is terminated.

The **DeleteDC** function removes the compatible display context since the application no longer needs it. Also, since the display context is no longer needed, the **ReleaseDC** function is used to release it. The Copy command is disabled and the variables used to identify the selection are set to zero. In other words, the copying operation removes the selection.

You could extend this example to incorporate a Cut command. A Cut command carries out the same action as the Copy command but also deletes the contents of the selection from the screen. The easy way to incorporate this command is to incorporate a Clear command, too, then have the Cut command carry out a copying and clearing operation.

13.3 Special Clipboard Topics

The clipboard provides a number of special features that an application can use to improve the usability of the clipboard and save itself some work. In particular, the clipboard lets applications delay the formatting of data passed to the clipboard until the data are needed, and lets applications draw within the Clipboard application's client area. Delaying formatting of data can save an application much time if the format is complex and no other application is likely to use it. Drawing in the Clipboard application's window lets an application display data formats that Clipboard does not know how to display. The following sections describe these features in more detail.

13.3.1 The Clipboard Application

The Clipboard application, *clipbrd.exe*, provides a way for the user to view the contents of the clipboard; for this reason, it is also known as the "clipboard viewer." It lists the names of all the formats for which handles (NULL or otherwise) exist in the clipboard, and displays the contents of the clipboard in one of these formats.

The clipboard viewer can display all the standard data formats. If there are handles for more than one standard data format, the clipboard viewer displays only one format, choosing from the following list, in decreasing order of priority: CF_TEXT, CF_METAFILEPICT, CF_BITMAP, CF_SYLK, and CF_DIF.

13.3.2 Rendering Data on Request

Applications that use many data formats can save formatting time by passing NULL data handles to the **SetClipboardData** function instead of generating all the data handles when a Cut or Copy command is used. The application does not actually have to generate a handle to the data until another application requests a handle by calling the **GetClipboardData** function.

When the **GetClipboardData** function is called with a request for a format that a NULL data handle has been set for, a WM_RENDERFORMAT message is sent to the clipboard owner. When an application receives this message, it can do the following:

1. Format the data last copied to the clipboard (the wParam value of WM_RENDERFORMAT specifies the format being requested).
2. Allocate a global memory block and copy the formatted data to it.
3. Pass the global memory handle and the format number to the clipboard by using the **SetClipboardData** function.

In order to accomplish these steps, the application needs to keep a record of the last data copied to the clipboard. The application may get rid of this information when it receives the WM_DESTROYCLIPBOARD message, which is sent to the clipboard owner whenever the clipboard is emptied by a call to the **EmptyClipboard** function.

13.3.3 Rendering Formats Before Terminating

When an application is destroyed, its knowledge of how to render data it has copied to the clipboard is also destroyed. Accordingly, a special message, WM_RENDERALLFORMATS, is sent when the application that owns the clipboard is being destroyed. Upon receiving this message, an

application should follow the steps described in Section 13.3.2, "Rendering Data on Request," for all formats that the application is capable of generating.

13.3.4 Registering Private Formats

In addition, an application may create and use private formats, or even new public ones. To create and use a new data-interchange format, an application must do the following:

1. Call the **RegisterClipboardFormat** function to register the name of the new format.
2. Use the value returned by **RegisterClipboardFormat** as the code for the new format when calling the **SetClipboardData** function.

Registering the format name ensures that the application is using a unique format number. In addition, it allows the Clipboard application to display the correct name of the data being held in the clipboard. For more information about displaying private data types in Clipboard, see Section 13.3.5, "Controlling Data Display in the Clipboard."

If two or more applications register formats with the same name, they will all receive the same format code. This allows applications to create their own public data types. If two or more applications register a format called WORKSHEET, for example, they will all have the same format number when calling the **SetClipboardData** and **GetClipboardData** functions, and will have a common basis for transferring WORKSHEET data between them.

13.3.5 Controlling Data Display in the Clipboard

There are two reasons why an application might wish to control the display of information in the Clipboard application. First, the application may have a private data type that is difficult or impossible to display in a meaningful way. Second, it may have a private data type that requires special knowledge to display.

13.3.5.1 Using a Display Format for Private Data

You can use a "display format" to represent a private data format that would otherwise be difficult or impossible to display. The data associated with display formats are text, bitmaps, or metafile pictures that the clipboard viewer can display as substitutes for the corresponding private data. To use a display format, you copy both the private data and the display data to the clipboard. When the clipboard viewer chooses a format to display, it chooses the display format instead of the private data.

There are three display formats: CF_DSPTEXT, CF_DSPBITMAP, and CF_DSPMETAFILEPICT. The data associated with these formats are identical to the text, bitmap, and metafile-picture formats. Since text, bitmaps, and metafile pictures are also standard formats, the clipboard viewer can display them without help from the application.

The following description assumes that the application has already followed the steps described in Section 13.2.1, "Copying Text to the Clipboard," to take ownership of the clipboard and set data handles.

To force the display of a private data type in a standard data format, the application must take the following steps:

1. Open the clipboard for alteration by calling the **OpenClipboard** function.
2. Create a global handle that contains text, a bitmap, or a metafile picture, specifying the information that should be displayed in the clipboard viewer.
3. Set the handle to the clipboard by calling the **SetClipboardData** function. The format code passed should be CF_DSPTEXT if the handle is to text, CF_DSPBITMAP if the handle is for a bitmap, and CF_DSPMETAFILEPICT if it is for a metafile picture.
4. Signal that it is done altering the clipboard by calling the **CloseClipboard** function.

13.3.5.2 Taking Full Control of the Clipboard-Viewer Display

An application can take complete control of the display and scrolling of information in the clipboard viewer. This control is useful when the application has a sophisticated private data type that only it knows how to display. Microsoft Write uses this facility for displaying formatted text.

The following description assumes that the application has already followed the steps described in Section 13.2.1, "Copying Text to the Clipboard," to take ownership of the clipboard and set data handles.

To take control of the display of information in the clipboard viewer, the application must take the following steps:

1. Open the clipboard for alteration by calling the **OpenClipboard** function.
2. Call the **SetClipboardData** function, using CF_OWNERDISPLAY as the data format, with a NULL handle.

3. Signal that it is done altering the clipboard by calling the **CloseClipboard** function.

The clipboard owner will then receive special messages associated with the display of information in the clipboard viewer. These messages are described in the following list:

Message	Action
WM_PAINTCLIPBOARD	Paint the specified portion of the window.
WM_SIZECLIPBOARD	Take note of the window size change.
WM_VSCROLLCLIPBOARD	Scroll the window vertically.
WM_HSCROLLCLIPBOARD	Scroll the window horizontally.
WM_ASKCBFORMATNAME	Supply the name of the displayed format.

You'll find full descriptions of these messages in the *Microsoft Windows Programmer's Reference.*

13.3.5.3 Using the Clipboard-Viewer Chain

The chaining together of clipboard-viewer windows provides a way for applications to be notified whenever a change is made to the clipboard. The notification, in the form of a WM_DRAWCLIPBOARD message, is passed down the viewer chain whenever the **CloseClipboard** function is called. The recipient of the WM_DRAWCLIPBOARD message must determine the nature of the change (Empty, Set, etc.) by calling the **EnumClipboardFormats** function, the **GetClipboardData** function, and other functions, as desired.

Any window that has made itself a link in the viewer chain must be prepared to do the following:

- Remove itself from the chain before it is destroyed.
- Pass along WM_DRAWCLIPBOARD messages to the next link in the chain.

The code for this action looks like this:

```
case WM_DESTROY:
    ChangeClipboardChain(hwnd, my_save_next);

    /* rest of processing for WM_DESTROY */

    break;
```

```
case WM_DRAWCLIPBOARD:
    if (my_save_next != NULL)
        SendMessage(my_save_next, WM_DRAWCLIPBOARD, wParam, lParam);

    /* rest of processing for WM_DRAWCLIPBOARD */

    break;
```

The my_save_next string is the value returned from the **SetClipboardViewer** function. These clipboard-viewer chain actions should be the first steps taken by the switch-statement branches that process the WM_DESTROY and WM_DRAWCLIPBOARD messages.

13.4 A Sample Application: ClipText

This sample application illustrates how to copy and paste from the clipboard. To create the ClipText application, copy and rename the source files of the EditMenu application, then make the following modifications:

1. Add new variables.
2. Add a WM_INITMENU case.
3. Modify the WM_COMMAND case to process the ID_COPY and ID_PASTE cases.
4. Compile and link the application.

This sample uses global memory to store the text to be copied. For a full explanation of global memory, see Appendix B, "Memory Management."

13.4.1 Add New Variables

You need to add several new global variables to hold the handles used for the copy and paste operations, as well as hold the addresses of the text strings. Add the following to the beginning of your C-language source file:

```
char a_string[] = "Hello Windows!";
HANDLE hData, hClipData;
LPSTR lpData, lpClipData;
```

The a_string array holds the string to be copied to the clipboard. The hData and hClipData variables hold the data handles to the global memory containing the copied text. The lpData and lpClipData pointer variables receive the addresses of the global memory blocks containing the strings.

13.4.2 Add a WM_INITMENU Case

You need to add a WM_INITMENU case to your window function to prepare the Edit menu for pasting. In general, the Paste command should not be available unless there is selected text in the clipboard to paste. Add the following statements to the window function:

```
case WM_INITMENU:
    if (wParam == GetMenu(hWnd)) {
        wFormat = 0;
        if (OpenClipboard(hWnd)) {
            while ((wFormat = EnumClipboardFormats(wFormat))
                != 0 && wFormat != CF_TEXT);
            CloseClipboard();
            if (wFormat == CF_TEXT)
                EnableMenuItem(wParam, ID_PASTE, MF_ENABLED);
            else
                EnableMenuItem(wParam, ID_PASTE, MF_GRAYED);
        }
    }
    return (TRUE);
```

These statements process the WM_INITMENU message only if the specified menu is the menu bar. The **EnumClipboardFormats** function enumerates the formats currently in the clipboard. If the CF_TEXT format is found, the **EnableMenuItem** function enables the command. Otherwise, the Paste command is disabled.

13.4.3 Modify the WM_COMMAND Case

You need to modify the IDM_COPY and IDM_PASTE cases in the WM_COMMAND case to process the Edit-menu commands. The IDM_COPY case must create a global memory block, fill it with text, and copy the handle of the block to the clipboard. The IDM_PASTE case must retrieve a handle from the clipboard and write the text in the client area.

Replace the existing IDM_COPY statement with the following statements:

```
case ID_COPY:
    hData = GlobalAlloc(GMEM_MOVEABLE,
        (DWORD) _lstrlen(a_string));
    if (!hData)
        break;
    lpData = GlobalLock(hData);
    _lstrcpy(lpData, (LPSTR) a_string);
    GlobalUnlock(hData);
    if (OpenClipboard(hWnd)) {
        EmptyClipboard();
        SetClipboardData(CF_TEXT, hData);
        CloseClipboard();
    }
    hData = 0;
    return ((long) TRUE);
```

The **GlobalAlloc** function allocates the global memory block used to store the text string. The locally defined _lstrcpy function copies the string into the block after the handle has been locked by the **GlobalLock** function. The handle must be unlocked before copying the handle to the clipboard. The **EmptyClipboard** function is used to remove any existing data from the clipboard.

Replace the IDM_PASTE statement with the following statements:

```
case ID_PASTE:
    if (OpenClipboard(hWnd)) {
        hClipData = GetClipboardData(CF_TEXT);
        CloseClipboard();
        if (!hClipData)
            break;
        hDC = GetDC(hWnd);
        lpClipData = GlobalLock(hClipData);
        TextOut(hDC, 10, 10, lpClipData, _lstrlen(lpClipData));
        GlobalUnlock(hClipData);
        ReleaseDC(hWnd, hDC);
    }
    return (TRUE);
```

The **GetClipboardData** function returns a handle to a global memory block. The **GlobalLock** function locks this handle, returning the block address that is used in the **TextOut** function to write the text.

13.4.4 Compile and Link

No changes are required to the **make** file to recompile and link the ClipText application. After compiling and linking it, start Windows, the Clipboard application, and ClipText. Then, choose the Copy command in the Edit menu. You should see something like Figure 13.1:

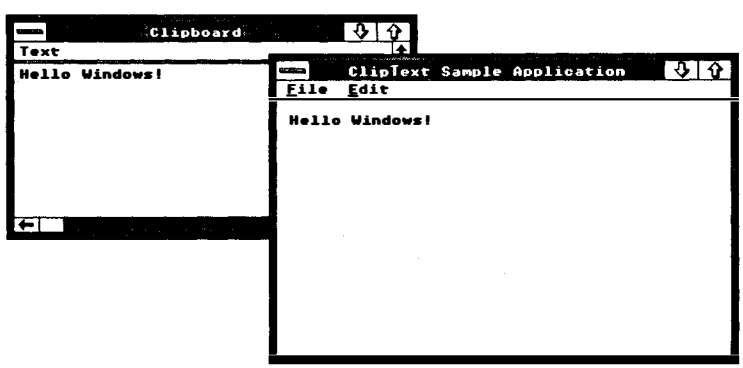


Figure 13.1 ClipText Window and Clipboard

13.5 A Sample Application: ClipBit

This sample application illustrates how to copy and paste bitmaps from the clipboard. To create the ClipBit application, copy and rename the source files of the ClipText application, then make the following modifications:

1. Add new variables.
2. Modify the WM_INITMENU case.
3. Modify the ID_COPY and ID_PASTE case.
4. Add the mouse cases to create a selection rectangle.
5. Compile and link the application.

This sample assumes that you have a mouse or other pointing device. It uses the selection functions provided by the Select library described in Appendix C, "Windows Libraries."

13.5.1 Add New Variables

You need a few new global variables to process the mouse-input messages. Add the following statements to the beginning of the C-language source file:

```
BOOL bTrack = FALSE;      /* TRUE if left button clicked */
RECT SelectRect;          /* Holds the current selection */
```

You also need some new local variables in the window function to copy or paste bitmaps. Add the following statements to the beginning of the window function:

```
HDC hMemoryDC;
HBITMAP hBitmap, hOldBitmap;
BITMAP PasteBitmap;
```

13.5.2 Modify the WM_INITMENU Case

You need to make two changes to the WM_INITMENU case. First, replace the CF_TEXT constant with CF_BITMAP. Second, use the **EnableMenuItem** function to disable the Copy command if the selection rectangle is empty. After you have changed it, the WM_INITMENU case should look like this:

```
case WM_INITMENU:
    if (wParam == GetMenu(hWnd)) {
        wFormat = 0;
```

```
        if (OpenClipboard(hWnd)) {
            while ((wFormat = EnumClipboardFormats(wFormat))
                != 0 && wFormat != CF_BITMAP);
            CloseClipboard();
            if (wFormat == CF_TEXT)
                EnableMenuItem(wParam, ID_PASTE, MF_ENABLED);
            else
                EnableMenuItem(wParam, ID_PASTE, MF_GRAYED);
        }
        if (IsRectEmpty(SelectRect))
            EnableMenuItem(wParam, ID_COPY, MF_GRAYED);
        else
            EnableMenuItem(wParam, ID_COPY, MF_ENABLED);
    }
    return (TRUE);
```

13.5.3 Modify the IDM_COPY and IDM_PASTE Cases

You need to modify the IDM_COPY and IDM_PASTE cases to process bitmaps. The IDM_COPY case must clear the selection rectangle from the client area, copy the bitmap to memory, and then copy the bitmap handle to the clipboard. The IDM_PASTE case must copy the handle into a memory device context, then copy the bitmap to the display.

After changes, the IDM_COPY case should look like this:

```
case ID_COPY:
    ClearSelection(hWnd, &SelectRect, SL_BLOCK);
    hMemoryDC = CreateCompatibleDC(hDC);
    if (hMemoryDC) {
        hBitmap = CreateCompatibleBitmap(hMemoryDC,
            SelectRect.right-SelectRect.left,
            SelectRect.bottom-SelectRect.top);
        if (hBitmap) {
            hOldBitmap = SelectObject(hMemoryDC, hBitmap);
            BitBlt(hMemoryDC, 0, 0,
                SelectRect.right - SelectRect.left,
                SelectRect.bottom - SelectRect.top,
                hDC, SelectRect.left, SelectRect.top, SRCCOPY);
            SelectObject(hMemoryDC, hOldBitmap);
            if (OpenClipboard(hWnd)) {
                EmptyClipboard();
                SetClipboardData(CF_BITMAP, hBitmap);
                CloseClipboard();
            }
        }
        DeleteDC(hMemoryDC);
    }
    ReleaseDC(hWnd, hDC);
    SetRectEmpty(SelectRect);
    break;
```

The **CreateCompatibleDC** function creates the memory device context needed to copy the selected bitmap from the client area to the clipboard. After a bitmap is created and selected, the **BitBlt** function copies the selected bitmap to the memory device context. Before copying the bitmap handle to the clipboard, the **SelectObject** function removes it from selection in the memory device context. After the copy operation is complete, the **DeleteDC** function deletes the memory device context since it is no longer needed. The **SetRectEmpty** function then clears the selection rectangle.

After you have changed it, the IDM_PASTE case should look like this:

```
case ID_PASTE:
    if (OpenClipboard(hWnd)) {
        hClipData = GetClipboardData(CF_BITMAP);
        CloseClipboard();
        if (!hClipData)
            break;
        hDC = GetDC(hWnd);
        hMemoryDC = CreateCompatibleDC(hDC);
        if (hMemoryDC) {
            GetObject(hClipData, sizeof(BITMAP),
                (LPSTR) & PasteBitmap);
            hOldBitmap = SelectObject(hMemoryDC, hClipData);
            if (hOldBitmap) {
                BitBlt(hDC, 10, 10,
                PasteBitmap.bmWidth, PasteBitmap.bmHeight,
                hMemoryDC, 0, 0, SRCCOPY);
                SelectObject(hMemoryDC, hOldBitmap);
            }
            DeleteDC(hMemoryDC);
       }
       ReleaseDC(hWnd, hDC);
    }
    break;
```

The **CreateCompatibleDC** function first creates the memory device context needed to display the bitmap retrieved from the clipboard. The **GetObject** function retrieves the bitmap dimensions since these are not provided by the clipboard. The **BitBlt** function then copies the bitmap to the client area.

13.5.4 Add the WM_LBUTTONDOWN, WM_MOUSEMOVE, and WM_LBUTTONUP Cases

You need to add the WM_LBUTTONDOWN, WM_MOUSEMOVE, and WM_LBUTTONUP cases to process the mouse input that creates the selection rectangle. Add the following statements to the window function:

```
case WM_LBUTTONDOWN:
    bTrack = TRUE;
       StartSelection(hWnd, MAKEPOINT(lParam), &SelectRect,
           (wParam & MK_SHIFT) ?
           SL_EXTEND | SL_BLOCK : SL_BLOCK);
    break;

case WM_MOUSEMOVE:
    if (bTrack)
        UpdateSelection(hWnd, MAKEPOINT(lParam),
            &SelectRect, SL_BLOCK);
    break;

case WM_LBUTTONUP:
    bTrack = FALSE;
    EndSelection(MAKEPOINT(lParam), &SelectRect);
    break;
```

These statements create a block selection. The selection remains in the client area until the user copies the contents of the selection to the clipboard.

13.5.5 Compile and Link

No changes are required to the **make** file to recompile and link the ClipBit application. After compiling and linking the application, start Windows, the Paint application, and ClipBit. Then, choose the Copy command in the Edit menu. You should see something like Figure 13.2:

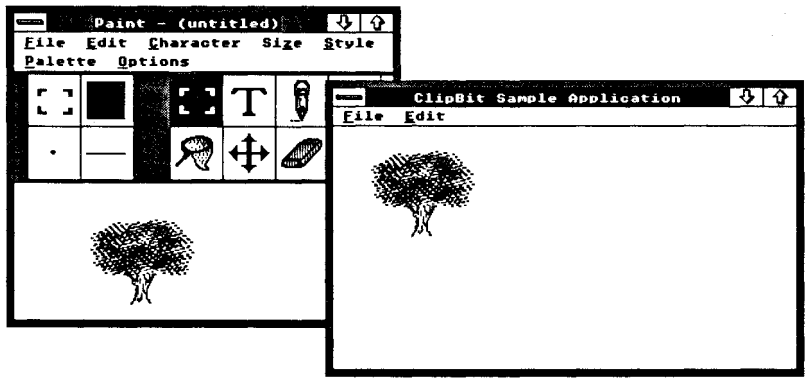


Figure 13.2 ClipBit Window

Appendixes

Appendix A
Fonts

A.1 Introduction

Microsoft Windows offers a rich array of text-writing capabilities that goes far beyond simple terminal-based text output. In particular, Windows lets you choose the font to be used for text output.

A font is a collection of characters that have a unique combination of height, width, typeface, character set, and other attributes. An application uses fonts to display or print text of varying faces and sizes. For example, a word-processing application uses fonts to give the user a "what you see is what you get" interface.

This appendix shows how to use fonts in your applications, and how to create font resources that your application and others can use.

A.2 Writing Text

You can write text in a given font by selecting the font and using the **TextOut** function to write it. **TextOut** writes the characters of the string by using the font that is currently selected in the device context. The following example shows how to write the string "This is a sample string":

```
hDC = GetDC(hWnd);
TextOut(hDC, 10, 10, "This is a sample string", 23);
ReleaseDC(hWnd, hDC);
```

In this example, **TextOut** starts the string at the coordinates (10,10) and prints all 23 characters of the string.

The default font for a device context is the system font. This is a fixed-width font representing characters in the ANSI character set. Its font name is "System." Windows uses the system font for menus, window captions, and other text.

A.3 Using Color when Writing Text

You can add color to the text you write by setting the text and background colors of the device context. The text color determines the color of the character to be written; the background color determines the color of everything in the character cell except the character. GDI writes the entire character cell (the rectangle enclosing the character) when it writes text. A character cell usually has the same width and height as the character.

You can set the text and background colors by using the **SetTextColor** and **SetBkColor** functions. The following example sets the text color to red and the background color to green:

```
SetTextColor(hDC, RGB(255,0,0));
SetBkColor(hDC, RGB(0,255,0));
```

When you first create a device context, the text color is black and the background color is white. You can change these colors at any time.

Note

If you are using a common display context, your colors are lost each time you release the context, so you need to set them each time you retrieve the display context.

The background color applies only when the background mode is opaque. The background mode determines whether the background color in the character cell has any affect on what is already on the display surface. If the mode is opaque, the background color overwrites anything already on the display surface; if it is transparent, anything on the display surface that would otherwise be overwritten by the background is preserved. You can set the background mode by using the **SetBkMode** function, or you can retrieve the current mode by using the **GetBkMode** function. Similarly, you can retrieve the current text and background color by using the **GetTextColor** and **GetBkColor** functions.

A.4 Using Stock Fonts

You are not limited to using the system font in your application. GDI offers a variety of stock fonts that you can retrieve and use as desired. To use stock fonts in your application, you must specify the type of font you want in the **GetStockObject** function. **GetStockObject** creates the font you request and returns a handle to the font that you can use to select into a device context. There are the following stock fonts:

Font	Description
SYSTEM_FONT	Specifies the system font. This is a fixed-pitch font based on the ANSI character set,

	and is used by the system to display window captions, menu names, and text in dialog boxes. The system font is always available. Other fonts are available only if they have been installed.
OEM_FIXED_FONT	Specifies a fixed-pitch font based on an OEM character set. OEM character sets vary from system to system. For IBM computers and compatibles, the OEM font is based on the IBM PC character set.
ANSI_FIXED_FONT	Specifies a fixed-pitch font based on the ANSI character set. For example, a Courier font is typically used, if one is available.
ANSI_VAR_FONT	Specifies a variable-width font based on the ANSI character set. For example, a Helvetica font is typically used, if it is available.
DEVICEDEFAULT_FONT	Specifies a font preferred by the given device. This font depends on how the GDI font mapper interprets font requests, so the font may vary widely from device to device.

To use a stock font, create it by using the **GetStockObject** function, then select the font handle into the device context by using the **SelectObject** function. Any subsequent calls to **TextOut** will use the selected font. The following example shows how to use the variable-width ANSI font:

```
HFONT hFont;
HFONT hOldFont;
    .
    .
    .

hFont = GetStockObject(ANSI_VAR_FONT);
if (hOldFont = SelectObject(hDC, hFont)) {
    TextOut(hDC, 10, 10, "This is a sample string", 23);
    SelectObject(hDC, hOldFont);
}
```

As you would with any other GDI object, you must select a font before it can be used in an output operation. The **SelectObject** function selects the font you have created and returns a handle to the previous font. The system stock font is always available, even if no other stock font is. If no other stock fonts are available, **GetStockObject** returns a handle to the system font.

A.5 Creating a Logical Font

A logical font is a list of font attributes, such as height, width, character set, and typeface, that you want GDI to consider when choosing a font for writing text. You can create a logical font by using the **CreateFont** function. **CreateFont** returns a handle to the logical font. You can use this handle in the **SelectObject** function to select the font for the device context. When you select a logical font, GDI chooses a physical font, based on your request, to write subsequent text. GDI attempts to choose a physical font that matches your logical font exactly, but if it cannot find an exact match in its internal pool of fonts, it chooses the closest matching font.

In the following example, the **CreateFont** function creates a logical font:

```
hFont = CreateFont(
    10,                             /* lfHeight           */
    8,                              /* lfWidth            */
    0,                              /* lfEscapement       */
    0,                              /* lfOrientation      */
    FW_NORMAL,                      /* lfWeight           */
    FALSE,                          /* lfItalic           */
    FALSE,                          /* lfUnderline        */
    FALSE,                          /* lfStrikeOut        */
    ANSI_CHARSET,                   /* lfCharSet          */
    OUT_DEFAULT_PRECIS,             /* lfOutPrecision     */
    CLIP_DEFAULT_PRECIS,            /* lfClipPrecision    */
    DEFAULT_QUALITY,                /* lfQuality          */
    FIXED_PITCH | FF_MODERN,        /* lfPitchAndFamily   */
    "System"                        /* lfFaceName         */
    );
```

This logical font asks for a fixed-pitch font in which each character is ten pixels high and eight pixels wide. Font dimensions are always described in pixels. The requested escapement and orientation are zero, which means the baseline along which the characters are displayed is horizontal and none of the characters will be rotated. FW_NORMAL is the requested weight. Other typical weights are, FW_BOLD (for darker, heavier characters) and FW_LIGHT (for lighter characters). Italic, underlined, or struckout characters are not desired. The requested character set is ANSI, the standard character set of Windows. Default output precision, clipping precision, and quality are requested. These attributes affect the way the characters are displayed. Setting these attributes to default values lets the display device take advantage of its own capabilities to display characters. The requested font family is FF_MODERN. The font name is "System".

When you supply a logical font to **SelectObject**, the function examines the pool of available fonts to find a font that satisfies the requested attributes. If it finds an exact match, it returns a handle to that font. If it fails to find an exact match, it chooses the closest possible font and returns that handle. In some cases, **SelectObject** may not find an exact match but nevertheless can synthesize the requested font by using an existing font that is close. For example, if the only available system font were five

pixels high and your logical font specified a height of ten pixels, **SelectObject** could synthesize the requested font by doubling the height. In such cases, **SelectObject** returns the synthesized font for writing text.

A.6 Using Multiple Fonts in a Line

If you are developing an application that uses a variety of fonts—a word processor, for instance—you will probably want to use more than one font in a line of text. To do so, you will need to write the text in each font separately. The **TextOut** function cannot change fonts for you.

The main difficulty with using more than one font in a line of text is that you need to keep track of how far each call to **TextOut** advances the line of text, so that you can supply the appropriate starting location for the next part of the line. If you are using variable-width fonts, keeping track of the length of a written string can be difficult. However, Windows provides the **GetTextExtent** function, which computes the length of a given string by using the widths of characters in the current font.

One way to write a line of text that contains multiple fonts is to use the **GetTextExtent** function after each call to **TextOut** and add the length to a current position. The following example shows how to write the line "This is a sample string.", using italic characters for the word "sample", and bold characters for all others:

```
X = 10;
SelectObject(hDC, hBoldFont);
TextOut(hDC, X, 10, "This is a ", 10);

X = X + LOWORD(GetTextExtent(hDC, "This is a ", 10));
SelectObject(hDC, hItalicFont);
TextOut(hDC, X, 10, "sample ", 7);

X = X + LOWORD(GetTextExtent(hDC, "sample ", 7));
SelectObject(hDC, hBoldFont);
TextOut(hDC, X, 10, "string.", 7);
```

In this example, the **SelectObject** function sets the font to be used in the subsequent **TextOut** function. The hBoldFont and hItalicFont font handles are assumed to have been previously created using the **CreateFont** function. Each **TextOut** function writes a part of the line, then the **GetTextExtent** function computes the length of that part. **GetTextExtent** returns a double-word value containing both the length and height. You need to use the **LOWORD** utility to retrieve the length. This length is added to the current position to determine the starting location of the next part of the line.

Another way to write a line with multiple fonts is to create a function that consolidates all the required actions into a single call. The following example shows such a function:

```
WORD StringOut(hDC, X, Y, lpString, hFont)
HDC hDC;
short X;
short Y;
LPSTR lpString;
HANDLE hFont;
{
    HANDLE hPrevFont;

    hPrevFont = SelectObject(hDC, hFont);
    TextOut(hDC, X, Y, lpString, _lstrlen(lpString));
    SelectObject(hDC, hPrevFont);
    return (LOWORD(GetTextExtent(hDC, lpString, _lstrlen(lpString))));
}
```

This function writes the string in the given font, then resets the font to its previous setting and returns the length of the written string. The following example shows how to write the line, "This is a sample string.":

```
X = 10;
X = X + StringOut(hDC, X, 10, "This is a ", hBoldFont);
X = X + StringOut(hDC, X, 10, "sample  ", hItalicFont);
StringOut(hDC, X, 10, "string.", hBoldFont);
```

A.7 Getting Information About the Selected Font

You can retrieve information about the selected font from a device context by using the **GetTextMetrics** and **GetTextFace** functions.

The **GetTextMetrics** function copies a **TEXTMETRIC** structure into a buffer that you supply. The structure contains a description of the font, including the average dimensions of the character cells within the font, the number of characters in the font, and the character set on which the font is based. You can use the text metrics to determine how much space you'll need between lines of text, or which character values have corresponding characters and which are represented by the font's default character.

The text metrics are most often used to determine how much space you need between lines of text to prevent one line from overwriting another. For example, to compute an appropriate value for single-line spacing, you add the values of the **tmHeight** and **tmExternalLeading** fields of the **TEXTMETRIC** structure. The **tmHeight** field specifies the height of each character cell and **tmExternalLeading** specifies the font designer's recommended spacing between the bottom of one character cell and the

top of the next. The following example shows how to write several lines with single-spacing:

```
TEXTMETRIC TextMetric;
int nLineSpacing;
int i;
    .
    .
    .
GetTextMetrics(hDC, &TextMetric);
nLineSpace = TextMetric.tmHeight + TextMetric.tmExternalLeading;

Y = 0;
for (i = 0; i < 4; i++) {
    TextOut(hDC, 0, Y, "Single-line spacing", 19);
    Y += nLineSpace;
}
```

You can also use the text metrics to verify that the selected font has the characteristics you need, such as weight, character set, pitch, and family. This is useful if you did not prepare the device context; for example, if you received it as part of a window message from a child window or control. For more information about the fields of the **TEXTMETRIC** structure, see the *Microsoft Windows Programmer's Reference.*

The **GetTextFace** function copies a name identifying the typeface of the selected font into a buffer that you supply. The name of the typeface together with the text metrics let you fully specify the font. The following example copies the name of the current font into the character array, FaceName.

```
char FaceName[32];
    .
    .
    .
GetTextFace(hDC, 32, FaceName);
```

A.8 Getting Information About a Logical Font

You can retrieve information about a font from the font handle by using the **GetObject** function. The **GetObject** function copies logical-font information, such as the height, width, weight, and character set, to a structure that you supply. You can use the logical-font information to see if the given font meets your needs. **GetObject** is often used after creating a font with the **CreateFont** function to see how closely the font matches the requested font. In the following example, **GetObject** retrieves logical-font information for a newly created font and compares the character-set values and facenames:

```
HFONT hFont;
LOGFONT LogFont;
    .
    .
    .
hFont = CreateFont(
    10,                                 /*          Height */
    10,                                 /*           Width */
    0,                                  /*      Escapement */
    0,                                  /*     Orientation */
    FW_NORMAL,                          /*          Weight */
    FALSE,                              /*          Italic */
    FALSE,                              /*       Underline */
    FALSE,                              /*       StrikeOut */
    OEM_CHARSET,                        /*         CharSet */
    OUT_DEFAULT_PRECIS,                 /*    OutPrecision */
    CLIP_DEFAULT_PRECIS,                /*   ClipPrecision */
    DEFAULT_QUALITY,                    /*         Quality */
    FIXED_PITCH | FF_MODERN,            /* PitchAndFamily */
    "Courier",                          /*        Typeface */
        );

GetObject(hFont, (LPLOGFONT) &LogFont);

if (LogFont.lfCharSet != OEM_CHARSET) {
    .
    .
    .
}
if (strcmp(LogFont.lfFaceName, "Courier")) {
    .
    .
    .
}
```

The font that GDI uses when you actually select a font by using the **SelectObject** function may vary widely from system to system. The selected font, which depends on the fonts available at the time of the selection, may or may not closely match your request. The only way to guarantee a request is to determine which fonts are actually available and request only those fonts, or add the appropriate font resource to the system font table before making the request, or change the method the font mapper uses to choose a font.

A.9 Enumerating Fonts

You can find out which fonts are available for a given device by using the **EnumFonts** function. This function sends information about the available fonts to a callback function that you supply. The callback function receives both logical-font and text-metric information. From this information you can determine which fonts you want to use and create appropriate font handles for them. If you create font handles by using the supplied information, you are guaranteed to get an exact match for the font when you select it for writing text.

The **EnumFonts** function usually provides font information about all the fonts that have a specific typeface name. You can supply the name when you call **EnumFonts**. If you do not supply a name, **EnumFonts** supplies information about arbitrarily selected fonts, each representing a typeface currently available. The way to examine all available fonts is to get a list of the available typefaces, then examine each font in each typeface.

The following example shows how to use **EnumFonts** to find out how many fonts having the "Courier" typeface are available. The callback function, EnumFunc, receives the font information and creates handles for each font:

```
FARPROC lpEnumFunc;
    .
    .
    .

int FAR PASCAL EnumFunc( )
{
}

hDC = GetDC(hWnd);
lpEnumFunc = MakeProcInstance(EnumFunc, hInst);
EnumFonts(hDC, "Courier", lpEnumFunc, NULL);
FreeProcInstance(lpEnumFunc);
```

To use the **EnumFonts** function, you must supply a callback function. As with all callback functions, EnumFunc must be explicitly named in the **EXPORTS** statement in your module-definition file and must be declared with the **FAR** and **PASCAL** attributes. For each font to be enumerated, the EnumFunc callback function receives a pointer to a logical-font structure, a pointer to a text-metrics structure, a pointer to any data you may have passed in the **EnumFonts** function call, and an integer specifying the font type. The following example shows a simple callback function that creates a list of all the sizes (in terms of height) of a given set of raster fonts:

```
short SizeList[10];
short SizeCnt = 0;
    .
    .
    .

int FAR PASCAL EnumFunc(lpLogFont, lpTextMetric, FontType, lpData)
LPLOGFONT lpLogFont;
LPTEXTMETRIC lpTextMetric;
short FontType;
LPSTR lpData;
{
    if (FontType & RASTER_FONTTYPE) {
         SizeList[SizeCnt++] = lpLogFont->lfHeight;
         if (SizeCnt >= 10)
             return (0);
    }
    return (1);
}
```

This example first checks the font to make sure it is a raster font. If the RASTER_FONTTYPE bit is 1, the font is a raster font; otherwise, it is a vector font. The next step is to save the value of the **lfHeight** field in the SizeList array. The callback function saves the first 10 sizes, then returns zero to stop the enumeration.

You can also use the DEVICE_FONTTYPE bit of the FontType parameter to distinguish GDI-supplied fonts from device-supplied fonts. This is useful if you want GDI to simulate bold, italic, underline, and strikeout attributes. GDI can simulate these attributes for GDI-supplied fonts, but not for device-supplied fonts.

A.10 Checking a Device's Text Capabilities

You can determine the extent of a device's text-writing capabilities by using the **GetDeviceCaps** function and the TEXTCAPS index. This index directs the function to return a bit flag identifying the text capabilities of the device. For example, you can use the text-capability flag to determine if the given device can use vector fonts, rotate characters, or simulate font attributes such as underlining and italicizing.

Most of the text capabilities apply to fonts that are supplied by the device as opposed to those supplied by GDI. Typically, GDI can scale fonts and simulate attributes for the fonts it supplies, but it cannot do so for device-supplied fonts. You can determine how many device fonts there are by using the **GetDeviceCaps** function with the NUMFONTS index. You can retrieve information about the device fonts by using the **EnumFonts** function and checking the DEVICE_FONTTYPE bit in the FontType parameter each time your **EnumFonts** callback function is called.

The following example shows how to make a list of device-supplied fonts. The **GetDeviceCaps** function returns the number of device-supplied fonts and **EnumFonts** creates font handles for each font:

```
HDC hDC;
HANDLE hDevFonts;
FARPROC lpEnumFunc;
short NumFonts;
    .
    .
    .
int FAR PASCAL EnumFunc(lpLogFont, lpTextMetric, FontType, Data)
LPLOGFONT lpLogFont;
LPTEXTMETRIC lpTextMetric;
short FontType;
LONG Data;
{
    PSTR pDevFonts;
    short index;
    int code = 1;
```

```
    if (FontType & DEVICE_FONTTYPE) {
        pDevFonts = LocalLock(LOWORD(Data));
        if (pDevFonts != NULL) {
            index = ++pDevFonts[0];
            if (index < HIWORD(Data))
                pDevFonts[index] = CreateFontIndirect(lpLogFont);
            else
                code = 0;
        }
        LocalUnlock(LOWORD(Data));
    }
    return (code);
}


    .
    .
    .
hDC = GetDC(hWnd);
NumFonts = GetDeviceCaps(hDC, NUMFONTS);
hDevFonts = LocalAlloc(LMEM_FIXED | LMEM_ZEROINIT,
    sizeof(HANDLE)*(NumFonts + 1));
lpEnumFunc = MakeProcInstance(EnumFunc, hInst);
EnumFonts(hDC, NULL, lpEnumFunc, MAKELONG(hDevFonts, NumFonts));
FreeProcInstance(lpEnumFunc);
```

A.11 Adding a Font Resource

GDI keeps a system font table that contains all the fonts that applications can use. GDI chooses a font from this table when an application makes a request for a font by using the **CreateFont** function.

A font resource is a group of individual fonts representing characters in a given character set that have various combinations of heights, widths, and pitches. For example, the system font resource contains a collection of fonts representing different sizes of characters in the ANSI character set. Similarly, the OEM font resource contains a collection of fonts representing different sizes of characters in an OEM character set.

An application can have up to 253 entries in the system font table.

Applications can load font resources and add the fonts in the resource to the system font table by using the **AddFontResource** function. Once a font resource is added, the individual fonts in the resource are accessible to the application. In other words, the **CreateFont** function considers the fonts when it tries to match a physical font with the specified logical font. (Fonts in the system font table are never directly accessible to an application. They are available only through the **CreateFontIndirect** or **CreateFont** functions, which return handles to the fonts, not memory addresses.)

You can add a font resource to the system font table by using the **AddFontResource** function. Similarly, to make room for other font resources, you can remove a font resource from the system font table by using the **RemoveFontResource** function.

Whenever an application adds or removes a font resource, it should inform all other applications of the change by sending a WM_FONTCHANGE message to the applications. You can use the following call to the **SendMessage** function to send the message to all windows:

```
SendMessage(-1, WM_FONTCHANGE, 0, 0L);
```

If the user has installed fonts by using the Control Panel application, you can retrieve a list of those fonts by using the **GetProfileString** function to search the [fonts] section of the *win.ini* file.

A.12 Setting the Text Alignment

The **TextOut** function uses a device context's current text alignment to determine how to position text relative to a given location. For example, the default text alignment is top-left, so **TextOut** places the upper-left corner of the character cell of the first character in the string at the specified location. That is, a function call such as the following places the upper-left corner of the letter "A" at the coordinates (10,10):

```
TextOut(hDC, 10, 10, "ABCDEF", 6);
```

You can change the text alignment for a device context by using the **SetTextAlign** function. If you think of **TextOut** as filling a rectangle with a text string, then you can think of the text alignment as specifying what part of the rectangle to place the specified point of the string in. **SetTextAlign** recognizes the left end, the center, and the right end of the rectangle, as well as the rectangle's top and bottom and the baseline within it. You can combine any one horizontal position with one vertical position to specify several combinations of alignment. For example, the following function sets the text alignment to right-bottom:

```
SetTextAlign(hDC, TA_RIGHT | TA_BOTTOM);
TextOut(hDC, 10, 10, "ABCDEF", 6);
```

This example places the lower-right corner of the letter "F" at the coordinates (10,10).

You can always determine the current text alignment by using the **GetTextAlign** function.

A.13 Creating Font-Resource Files

You can create your own font resources for your application and others by creating font files and adding them as resources to a font-resource file. To create a font-resource file, you must follow these steps:

1. Create the font files.
2. Create a font-resource script.
3. Create a dummy code module.
4. Create a module-definition file that describes the fonts and the devices that use the fonts.
5. Compile and link the sources.

A font-resource file is actually an empty Windows library—it contains no code or data, but does contain resources. Once you have font files, you can add them to the empty library by using the resource compiler. You can also add other resources to the library, such as icons, cursors, and menus.

Note

You should not add fonts to an application's resources. These resources should be reserved for use by the application only.

The following sections explain how to create font-resource files.

A.13.1 Creating Font Files

Before creating a font resource, you need one or more font files. You can create font files by using the Windows 2.0 Font Editor, described in *Microsoft Windows Programming Tools*. You can also create original files by using the font-file format given in the *Microsoft Windows Programmer's Reference*. You are free to determine the number, size, and type of font files in a font resource. In most cases, you should include enough fonts to reasonably satisfy most logical-font requests for the device the fonts are to be used with.

When planning font sizes, remember that GDI can scale device-independent raster fonts by 1 to 8 times vertically and 1 to 5 times horizontally. GDI can also simulate bold, underlined, strikeout, and italic fonts. Although scaled or simulated fonts do not look as nice as actual fonts, they can save valuable memory resources.

A.13.2 Creating the Font-Resource Script

You add the resources to the file by adding one or more **FONT** statements to your resource script file. The statement has the following form:

number **FONT** *filename*

One statement is required for each font file to be placed in the resource. The *number* must be unique since it is used to identify the font later. The following is a typical resource script file for a font resource:

```
1  FONT FntFil01.FNT
2  FONT FntFil02.FNT
3  FONT FntFil03.FNT
4  FONT FntFil04.FNT
5  FONT FntFil05.FNT
6  FONT FntFil06.FNT
```

Fonts can be added to modules that contain other resources by adding them to the existing resource script. This means you can have icon, menu, cursor, and dialog-box definitions in the resource script file, as well as in **FONT** statements.

Note

To avoid loading unneeded fonts, it is recommended that each font resource contain fonts that represent characters designed for only one aspect ratio or resolution.

A.13.3 Creating the Dummy Code Module

The dummy code module provides the object file from which the font-resource file is made. You create the dummy code module by using the assembler and the Cmacros. The module's source file should like like this:

```
TITLE   FONTRES - Stub file to build FONTRES.EXE

.xlist
include cmacros.inc
.list

sBegin  CODE
sEnd    CODE
end
```

Assemble this source file by using the **masm** command. It will create an object file that contains no code and no data, but which can be linked to an empty Windows library to which you can add the font resources.

A.13.4 Creating the Module-Definition File

You need to create a module-definition file for the font resource. The file must contain a **LIBRARY** statement defining the resource name, a **DESCRIPTION** statement that describes the font-resource characteristics, and a **DATA** statement. The module-definition file for a font resource should look like this:

```
LIBRARY FontRes

DESCRIPTION 'FONTRES 133,96,72 : System, Terminal (Set #3)'

DATA NONE
```

The **DESCRIPTION** statement provides device-specific information about the font that can be used to match a font with a given display or printer. The following are the three possible formats for the **DESCRIPTION** statement in a font resource:

DESCRIPTION 'FONTRES *Aspect, LogPixelsX, LogPixelsY*: *Cmt*'

DESCRIPTION 'FONTRES CONTINUOUSSCALING: *Cmt*'

DESCRIPTION 'FONTRES DEVICESPECIFIC *DeviceTypeGroup*: *Cmt*'

The first format specifies a font that was designed for a specific aspect ratio and logical pixel width and height, and can be used with any device having the same aspect and logical pixel dimensions. *Aspect* is the value (100**AspectY*)/*AspectX* rounded to an integer. The *AspectX*, *AspectY*, *LogPixelsX*, and *LogPixelsY* values are the same as given in the corresponding device's **GDIINFO** structure (the values of which are accessible by using the **GetDeviceCaps** function). You can give more than one set of *Aspect*, *LogPixelX*, and *LogPixelY* values, if desired. The *Cmt* value is a comment. The following statements are examples:

```
DESCRIPTION 'FONTRES 133,96,72: System, Terminal (Set #3)'
DESCRIPTION 'FONTRES 200,96,48; 133,96,72; 83,60,72; 167,120,72: Helv'
```

The second format specifies a continuous scaling font. This typically corresponds to vector fonts that can be drawn to any size and that do not depend on the aspect or logical pixel width of the output device. The following statement is an example:

```
DESCRIPTION 'FONTRES CONTINUOUSSCALING : Modern, Roman, Script'
```

The third format specifies a font that is specific to a particular device or group of devices. The *DeviceTypeList* can be **DISPLAY** or a list of device-type names, the same names you would specify as the second parameter in a call to the **CreateDC** function. For example:

```
DESCRIPTION 'FONTRES DISPLAY: HP 7470 plotters'
DESCRIPTION 'FONTRES DEVICESPECIFIC HP 7470A, HP 7475A: HP 7470 plotters'
```

Note

The maximum length of a **DESCRIPTION** line is 127 characters.

A.13.5 Compiling and Linking the Font-Resource File

The following **make** file lists the commands required to compile and link the font-resource file:

```
fontres.obj: fontres.asm
    masm fontres;

fontres.exe: fontres.def fontres.obj fontres.rc fontres.exe \
                FntFil01.FNT FntFil02.FNT FntFil03.FNT \
                FntFil04.FNT FntFil05.FNT FntFil06.FNT
    link4 fontres.obj, fontres.exe, NUL, /NOD, fontres.def
    rc fontres.rc
    rename fontres.exe fontres.fon
```

By convention, all font-resource files have the *.fon* filename extension. The last line in the **make** file renames the executable file to *fontres.fon*.

A.14 A Sample Application: ShowFont

This sample application illustrates how to use fonts in a Windows application. Although the ShowFont application has the same basic structure as any application described in this guide, it contains considerably more statements, in a far greater variety, than any other sample application. For this reason, a full description of the application is given in the application source files found on the Learning Guide Samples Disk.

The ShowFont application illustrates more than how to use fonts. It also shows how to modify many of the tasks previously described in this guide in order to carry out slightly different tasks. For example, it shows how to create and use modeless dialog boxes, how to use list boxes with your own strings (instead of the current directory), and how to use Windows' direct-access method for group boxes and radio buttons in a dialog box.

Appendix B
Memory Management

B.1 Introduction

This appendix provides tips for using the Microsoft Windows memory-management system. The system lets you allocate and manage extra memory for an application while it is running. Windows also uses the system to manage the code and data segments of your application.

B.2 Using Memory

Windows provides a memory-management system that lets you allocate blocks of memory for use in your applications. You can allocate blocks for memory from either the global or the local heap. The global heap is a pool of free memory available to all applications. The local heap is a pool of free memory available to just your application.

In most memory-management systems, the memory you allocate remains fixed at a specific memory location until you free it. In Windows, allocated memory can be movable and discardable, as well as fixed. A movable memory block does not have a fixed address; Windows can move it at any time to a new address. Movable memory blocks let Windows make best use of free memory. For example, if a movable memory block separates two free blocks of memory, Windows can move the movable block to combine the free blocks into one contiguous block. A discardable memory block is similar to movable memory in that windows can move it, but Windows can also reallocate a discardable block to zero length if it needs the space to satisfy an allocation request. Reallocating a memory block to zero length destroys the data the block contains, but an application always has the option of reloading the discarded data whenever it is needed.

When you allocate a memory block, you receive a handle to that block and not a pointer. The memory handle identifies the allocated block. You use it to retrieve the block's current address when you need to access the memory.

To access a memory block, you lock the memory handle. This temporarily fixes the memory block and returns a pointer to its beginning. While a memory handle is locked, Windows cannot move or discard the block. Therefore, after you have finished using the block, you should unlock the handle as soon as possible. Keeping a memory handle locked makes Windows' memory management less efficient and may cause subsequent allocation requests to fail.

Windows lets you compact memory. By squeezing the free memory from between allocated memory blocks, Windows collects the largest contiguous free-memory block possible, from which you may allocate additional blocks of memory. This squeezing is a process of moving and (if necessary)

discarding memory blocks. Windows also lets you discard individual memory blocks if you temporarily have no need for them.

B.2.1 Using the Global Heap

The global heap contains all of system memory. Windows allocates the memory it needs for code and data from the global heap when it first starts. Any remaining free memory in the global heap is available to applications and Windows libraries.

You can allocate any size of memory block from the global heap. Applications typically allocate large blocks from the global heap. These blocks can exceed 64 kilobytes (K) if the applications need that much contiguous space. As with all memory allocations, Windows returns a handle identifying the block when it is allocated. To use the block, the application must lock it. At this point, Windows returns a full 32-bit address to the first byte in the block.

You can allocate a block of global memory by using the **GlobalAlloc** function. You specify the size and type (fixed, movable, or discardable), and **GlobalAlloc** returns a handle to the block. Before you can use the memory block, you must lock it by using the **GlobalLock** function, which returns the full 32-bit address of the first byte in the memory block. You may then use this long pointer to access the bytes in the block. In the following example, the **GlobalAlloc** function allocates 4096 bytes of movable memory and the **GlobalLock** function locks it so that the first 256 bytes can be set to 0xFF:

```
HANDLE hMem;
LPSTR lpMem;
int i;

if ((hMem = GlobalAlloc(GMEM_MOVEABLE, 4096)) != NULL) {
    if ((lpMem = GlobalLock(hMem)) != (LPSTR) NULL) {
        for (i = 0; i < 256; i++)
            lpMem[i] = 0xFF;
        GlobalUnlock(hMem);
    }
}
```

In this example, the application unlocks the memory handle by using the **GlobalUnlock** function as soon as possible after accessing the memory block. Once a movable or discardable memory block is locked, Windows guarantees that the block will remain fixed in memory until it is unlocked. This means the address remains valid as long as the block remains locked, but this also keeps Windows from making the best use of memory if other allocation requests are made. Cooperative applications unlock memory.

The **GlobalAlloc** function returns NULL if an allocation request fails. You should always check the return value to ensure that a valid handle exists. If desired, you can check to see how much memory is available in the global heap by using the **GlobalCompact** function. This function returns the number of bytes in the largest contiguous free block of memory.

You should also check the address returned by the **GlobalLock** function. This function returns a null pointer if the memory handle was not valid or if the contents of the memory block have been discarded.

You can free any global memory you may no longer need by using the **GlobalFree** function. In general, you should free memory as soon as you no longer need it so that other applications can use the space. Although it is a good idea to free global memory before your application terminates, Windows will automatically free it if you do not.

B.2.2 Using the Local Heap

The local heap contains free memory that may be allocated for private use by the application. The local heap is located in the application's data segment and is therefore accessible only to a specific instance of the application. You can allocate memory from the local heap in blocks of up to 64K and the memory can be fixed, movable, or discardable, as needed.

Applications do not automatically have local heaps. You must specify a local heap for an application by using the **HEAPSIZE** statement in the application's module-definition file. The statement sets the initial size, in bytes, of the local heap. If the local heap is in a fixed data segment, you may allocate up to the specified heap size. If the local heap is in a movable data segment, you may allocate beyond the initial heap size and up to 64K, since Windows will automatically allocate additional space for the local heap until the data segment reaches the 64K maximum. You should note, however, that if Windows allocates additional local memory to satisfy a local allocation, it may move the data segment, making invalid any long pointers to blocks in local memory.

The maximum size of any local heap depends on the size of the application's stack and static and global data. The local heap shares the data segment with the stack and this data, but since a data segment may be no larger than 64K, the local heap for an application can be no larger than 64K less the size of the stack and the size of the application's global and static data. The application's stack size is defined by the **STACKSIZE** statement given in the application's module-definition file. The global and static data size depends on how many strings and global or static variables are declared in the application.

You can allocate local memory by using the **LocalAlloc** function. The function allocates a block of memory in the application's local heap and returns a handle to the memory. You lock the local memory block by using the **LocalLock** function. This returns a 16-bit offset to the first byte in the memory block. The offset is relative to the beginning of your data segment. In the following example, the **LocalAlloc** function allocates 256 bytes of movable memory and the **LocalLock** function locks it so that the first 256 bytes can be set to 0xFF:

```
HANDLE hMem;
PSTR pMem;
int i;

if ((hMem = LocalAlloc(LMEM_MOVEABLE, 256)) != NULL) {
    if ((pMem = LocalLock(hMem)) != NULL) {
        for (i = 0; i < 256; i++)
            pMem[i] = 0xFF;
        LocalUnlock(hMem);
    }
}
```

In this example, the application unlocks the memory handle by using the **LocalUnlock** function as soon as possible after accessing the memory block. Once a movable or discardable memory block is locked, Windows guarantees that the block will remain fixed in memory until it is unlocked. This means the address remains valid as long as the block remains locked, but this also keeps Windows from making best use of memory if other allocation requests are made. If you want to ensure that you are getting the best performance from your application's local heap, make sure you unlock memory after using it.

The **LocalAlloc** returns NULL if an allocation request fails. You should always check the return value to ensure that a valid handle exists. If desired, you can check to see how much memory is available in the global heap by using the **LocalCompact** function. This function returns the number of bytes in the largest contiguous free block of memory.

You should also check the address returned by the **GlobalLock** function. This function returns NULL if the memory handle was not valid or if the contents of the memory block have been discarded.

B.2.3 Working with Discardable Memory

You create a discardable memory block by combining the GMEM_DISCARDABLE and GMEM_MOVEABLE options when allocating the block. The resulting block will be moved as necessary to make room for other allocation requests, or if there is not enough memory to satisfy the request, the block may be discarded. The following example allocates a discardable block from global memory:

```
hMem = GlobalAlloc(GMEM_MOVEABLE | GMEM_DISCARDABLE, 4096L);
```

When Windows discards a memory block, it empties the block by reallocating it, with zero bytes given as the new size. The contents of the block are lost, but the memory handle to this block remains valid. Any attempt to lock the handle and access the block will fail, however.

Windows determines which memory blocks to discard by using a "least recently used" (LRU) algorithm. It continues to discard memory blocks until there is enough memory to satisfy an allocation request. In general, if you have not accessed a discardable block in some time, it is a candidate for discarding. A locked block cannot be discarded.

You can discard your own memory blocks by using the **GlobalDiscard** function. This function empties the block but preserves the memory handle. You can also discard other applications' memory blocks by using the **GlobalCompact** function. This function moves and discards memory blocks until the specified or largest possible amount of memory is available. One way to discard all discardable blocks is to supply –1 as the argument. This is a request for every byte of memory. Although the request will fail, it will discard all discardable blocks and leave the largest possible block of free memory.

Since a discarded memory block's handle remains valid, you can still retrieve information about the block by using the **GlobalFlags** function. This is useful for verifying that the block has actually been discarded. **GlobalFlags** sets the GMEM_DISCARDED bit in its return value when the specified memory block has been discarded. Therefore, if you attempt to lock a discardable block and the lock fails, you can check the block's status by using **GlobalFlags**.

Once a discardable block has been discarded, its contents are lost. If you wish to use the block again, you need to reallocate it to its appropriate size and fill it with the data it previously contained. You can reallocate it by using the **GlobalReAlloc** function. The following example checks the block's status, then fills it with data if it has been discarded:

```
lpMem = GlobalLock(hMem);

if (lpMem == (LPSTR) NULL) {
    if (GlobalFlags(hMem) & GMEM_DISCARDED) {
        hMem = GlobalReAlloc(hMem, 4096L,
            GMEM_MOVEABLE | GMEM_DISCARDABLE);
        lpMem = GlobalLock(hMem);

        /* Fill with data */
        GlobalUnlock(hMem);
    }
}
    .
    .
    .
```

You can make a discardable object nondiscardable (or vice versa) by using the **GlobalReAlloc** function and the GMEM_MODIFY flag as shown in the following example:

```
hMem = GlobalReAlloc(hMem, 4096L, GMEM_MODIFY | GMEM_MOVEABLE);
```

This example changes a discardable block, identified by the hMem parameter, to a movable block.

B.3 Using Segments

One of the principal features of Windows is that it lets the user run more than one application at a time. Since multiple applications place greater demands on memory than does a single application, Windows' ability to run more than one application at a time has a significant impact on how you write applications. Although many computers typically have at least 640K of memory, this memory rapidly becomes a limited resource as the user loads and runs more and more applications. In Windows, you must be conscious of how your application uses memory and be prepared to minimize the amount of memory your application occupies at any given time.

To help you manage your application's use of memory, Windows uses the same memory-management system for your application's code and data segments that you use within your application to allocate and manage global memory blocks. When the user starts your application, Windows allocates space for the code and data segments in global memory, then copies the segments from the executable file into memory. These segments can be fixed, movable, and even discardable. You specify their attributes in the application's module-definition file.

You can reduce the impact your application has on memory by using movable code and data segments. **MOVEABLE** is, in fact, the default attribute that code and data segments typically receive. Using movable segments lets Windows at least move the segments, when necessary, in order to take advantage of free memory as it becomes available. To prevent disaster, Windows moves a segment only if it is not "busy"; that is, if it is not currently executing or being accessed.

You can minimize your application's impact on memory by using discardable code segments. If you make a code segment discardable, Windows discards it, if necessary, to satisfy requests for global memory. Unlike ordinary memory blocks that you may allocate, discarded code segments are monitored by Windows, which automatically reloads them if your application attempts to execute code within the segment. This means that your application's code segments are in memory only when they are needed.

If you use discardable code segments, you must not store data in the segment. Discarding a segment destroys the contents of the segment. Windows does not save the current contents of a discarded segment. Instead it assumes that the segment is no different than when originally loaded and will load the segment directly from the executable file when it is needed.

B.3.1 Code Segments

A code segment is one or more bytes of machine instructions. It represents all or part of an application's program instructions. A code segment is never greater than 64K.

Every application has at least one code segment. For example, the sample applications described in previous chapters have one and only one code segment. You can also create an application that has multiple code segments. In fact, most Windows applications have multiple code segments. These segments let you reduce the size of any given code segment to just the number of instructions needed to carry out some task. If you also make these segments discardable, you effectively minimize the memory requirements of your application's code segments.

When you create medium- or large-model applications, you are creating applications that use multiple code segments. Medium- and large-model applications typically have one or more source files for each segment. You compile each source file separately and explicitly name the segment to which the compiled code will belong. Then you link the application, naming the segments and defining their attributes in the application's module-definition file.

To define a segment's attributes, you use the **SEGMENTS** statement in the module-definition file. The following example shows definitions for three segments:

```
SEGMENTS
        PAINT_TEXT MOVEABLE DISCARDABLE
        INIT_TEXT MOVEABLE DISCARDABLE
        WNDPROC_TEXT MOVEABLE DISCARDABLE
```

You may also use the **CODE** statement in the module-definition file to define the attributes of the default code segment, _TEXT. The C compiler typically creates a _TEXT segment for you when you create a small-model application. Also, C run-time code that the linker may append to your application is often in the _TEXT segment. The following example shows how to make the segment discardable:

```
CODE MOVEABLE DISCARDABLE
```

If you use discardable code segments in your application, you need to balance discarding with the number of times the segment may be accessed. For example, the segment containing your main window function probably should not be discardable since Windows calls the function often. Since a discarded segment has to be loaded from disk when needed, the savings in memory you may realize by discarding the window function may be offset by the loss in performance that comes with accessing the disk often.

B.3.2 The DATA Segment

Every application has a **DATA** segment. The **DATA** segment contains the application's stack, local heap, and static and global data. Like a code segment, the **DATA** segment cannot be larger than 64K.

A **DATA** segment can be fixed or movable, but not discardable. If the **DATA** segment is movable, Windows automatically locks the segment when it passes control to the application. Otherwise, a movable **DATA** segment may move if an application allocates global memory, or the application attempts to allocate more memory than is currently available in the local heap. For this reason, it is important not to keep long pointers to variables in the **DATA** segment.

You can define the attributes of the **DATA** segment by using the **DATA** statement in the module-definition file. The default attributes are movable and multiple. The multiple attribute directs Windows to create one copy of an application's data segment for each instance of the application. This means the contents of the **DATA** segment are unique to each instance of the application.

A large-model application may have additional data segments, but only one **DATA** segment. In Windows, any additional data segments must be explicitly defined in the **SEGMENTS** statement of the module-definition file and must be fixed.

B.4 A Sample Application: Memory

This sample application illustrates how to create a medium-model Windows application that uses discardable code segments. To create the Memory application, copy and rename the source files of the Generic application, then make the following modifications:

1. Split the C-language source file into four separate files.
2. Modify the include file.

3. Add new segment definitions to the module-definition.
4. Modify the **make** file.
5. Compile and link the application.

The following sections describe each step in detail.

B.4.1 Split the C-Language Source File

You need to split the C-language source file into separate files so that the functions within the file are compiled as separate segments. For this application, you can split the source file into four parts, as described in the following list:

Source File	Content
memory1.c	Contains the **WinMain** function. Since Windows executes **WinMain** fairly often, the segment created from this source file is not discardable. This is to prevent a situation in which the segment has to be loaded from the disk often. Since **WinMain** is relatively small anyway, keeping this segment in memory has very little impact on available global memory.
memory2.c	Contains the MemoryInit function. Since the MemoryInit function is used only when the application first starts, the segment created from this source file can be discardable.
memory3.c	Contains the MemoryWndProc function. Although the segment created from this source file can be discardable, the MemoryWndProc function is likely to be called at least as often as the **WinMain** function receives control. In this case, the segment is movable but not discardable.
memory4.c	Contains the About function. Since the About function is seldom called (only when the About dialog box is displayed), the code segment created from this source file can be discardable.

You must include the *windows.h* and *memory.h* files in each source file.

B.4.2 Modify the Include File

You need to move the declaration of the hInst variable into the *memory.h* file. This ensures that the variable is accessible in all segments. The hInst variable is used in the **WinMain** and MemoryWndProc functions.

B.4.3 Add New Segment Definitions

You need to add segment definitions to the module-definition file to specify the attributes of each code segment. This means you need to add a **SEGMENTS** statement to the file and list each segment by name in the application. After changes, the module-definition file should look like this:

```
;Module-definition file for Memory

NAME     Memory          ; application's module name

DESCRIPTION 'Sample Microsoft Windows Application'

STUB     'WINSTUB.EXE' ; Make sure it doesn't run without Windows

CODE     MOVEABLE        ; Code can be moved in memory

SEGMENTS
    MEMORY_MAIN PRELOAD MOVEABLE
    MEMORY_INIT LOADONCALL MOVEABLE DISCARDABLE
    MEMORY_WNDPROC LOADONCALL MOVEABLE
    MEMORY_ABOUT LOADONCALL MOVEABLE DISCARDABLE

;DATA must be MULTIPLE if program can be invoked more than once
DATA     MOVEABLE MULTIPLE

HEAPSIZE  1024
STACKSIZE 4096  ; recommended minimum for Windows applications

; All functions that will be called by any Windows routine
; must be exported.
EXPORTS
    MemoryWndProc    @1  ; name of window-processing function
    About            @2  ; name of About processing function
```

The **SEGMENTS** statement defines the attributes of each segment: the MEMORY_MAIN segment contains **WinMain**; MEMORY_INIT contains the MemoryInit function; MEMORY_WNDPROC contains the window function; and MEMORY_ABOUT contains the dialog function. Each segment has the **MOVEABLE** attribute, but only MEMORY_INIT and MEMORY_ABOUT have the **DISCARDABLE** attribute. Also, only the MEMORY_MAIN segment is loaded when the application is started. The other segments have the **LOADONCALL** attribute, which means they are loaded when needed.

Although each segment is explicitly defined, the **CODE** statement is still given. This statement specifies the attributes of any additional segments the linker may add to the application; for example, segments containing C run-time functions called in the application source files.

B.4.4 Modify the Make File

You need to modify the **make** file to separately compile the new C-language sources. Since this application is a medium-model application, you need to use the **–AM** option when compiling. For clarity, you should also name each segment by using the **–NT** option when compiling. After changes, the **make** file should look like this:

```
memory.res: memory.rc memory.h
    rc -r memory.rc

memory1.obj: memory1.c memory.h
    cl -c -AM -Gw -Zp -NT MEMORY_MAIN memory1.c

memory2.obj: memory2.c memory.h
    cl -c -AM -Gw -Zp -NT MEMORY_INIT memory2.c

memory3.obj: memory3.c memory.h
    cl -c -AM -Gw -Zp -NT MEMORY_WNDPROC memory3.c

memory4.obj: memory4.c memory.h
    cl -c -AM -Gw -Zp -NT MEMORY_ABOUT memory4.c

memory.exe: memory1.obj memory2.obj memory3.obj memory4.obj memory.def
    link4 memory1 memory2 memory3 memory4,memory.exe,,mlibw,memory.def
    rc memory.res

memory.exe: memory.res
    rc memory.res
```

You must link with the *mlibw.lib* library instead of the *slibw.lib* library.

B.4.5 Compile and Link

After compiling and linking the Memory application, start Windows, the Heapwalker application (provided with the Microsoft 2.0 Windows Software Development kit), and Memory. Use Heapwalker to view the various segments of the Memory application.

Appendix C
Windows Libraries

C.1 Introduction

A Microsoft Windows library is an executable module containing functions that Windows applications can dynamically link to and call in order to perform useful tasks. A Windows library is similar to run-time libraries, such as the C run-time library, except that it is linked with the application when the application is run, not when the application is linked with the linker. This method of linking a library with an application when the application runs is called dynamic linking.

Dynamic linking is an important extension of traditional linking methods. It makes linking more efficient by permitting applications to link with a single copy of a library. Only one copy of a library needs to be in memory. Applications using the library link to the same copy, unlike traditional linking, in which one copy of a library is required for each application.

All Windows libraries are dynamic-link libraries. For example, the *gdi.exe*, *user.exe*, and *kernel.exe* files that comprise the major part of Windows are dynamic-link libraries. You can create your own dynamic-link libraries for your applications by following the directions given in this appendix.

C.2 Creating a Library

You create Windows libraries much as you would Windows applications: you write the sources, then compile, link, and add the resources. However, many of the steps are different from those needed to create Windows applications. To make a Windows library, follow these steps:

1. Write the initialization function and the library functions.
2. Create a module-definition file that contains a **LIBRARY** statement naming the library and an **EXPORTS** statement for each function in the library to be exported.
3. Compile the library source file.
4. Use the **link4** command to create the *.exe* file.
5. Use the **rc** command to compile and add resources to the library's executable file.
6. Use the **implib** command to create an import library that can be linked with application source files that call functions in the Windows library.
7. Use a text editor to create an include file to be used in application source files that call the library functions. The include file should define each function's return and parameter types.

C.2.1 Create the Initialization Function

A Windows library can have an initialization function, if one is desired. Windows calls the function when it first loads the library, giving the function the opportunity to carry out any initialization tasks the library may need, such as initializing the local heap. The initialization function, when called, must return to Windows immediately after completing its tasks. Unlike the **WinMain** function in an application, the initialization function must not create windows or enter a message loop.

Windows does not require a library to have an initialization function. If a library does not require initialization, no function is required.

To create an initialization function, you will need to write it in assembly language. Windows does not use a high-level calling convention when calling this function. Instead, it passes relevant data in registers and expects the function to have direct access to these registers. When a library's initialization function is first called, the registers contain the following information:

Register	Contents
DI	Contains the module handle of the library.
DS	Contains the segment address of the library's data segment.
CX	Contains the size of the local heap. This is the same value given in the **HEAPSIZE** statement in the library's module-definition file.

The **CS**, **IP**, **SS**, and **SP** registers contain current addresses of code and stack. All other registers are free to use. No special values are passed in them.

To create an initialization function, you must do the following:

- Specify the initialization function's address as the library's entry point. You do this by naming the function in the **END** statement in the assembly-language file.
- Avoid excessive use of the stack. The initialization function uses the system stack, which is a limited resource.
- If the initialization is successful, return a nonzero value in the **AX** register. Otherwise, return zero in the **AX** register to direct Windows to cancel the load operation.

You can use the Cmacros to create the initialization file. The file should have the following form:

```
?WIN=1
include include
assumes cs,CODE

sBegin  CODE

cProc   LoadLib,<FAR,PUBLIC,NODATA>,<si,di>
cBegin
        xor     ax,ax                   ; Prepare error return value

         .                              ; Initialization code
         .
         .

LoadExit:
        mov ax, 1                       ; On success, return non-zero
LoadError:                              ; Otherwise, return zero
cEnd

sEnd    CODE

        end     LoadLib
```

In this example, the initialization function, **LoadLib**, prepares the error return value in the **AX** register, then carries out any initialization. Since the initialization function is constructed by using the Cmacros, it can call C-language helper functions that you have written for the library, or even call functions in other Windows libraries.

You can assemble the initialization file by using the **masm** command. No special options are required. You can then link the resulting object file with the object files containing the library's functions. A library's functions can be written in the C language, if desired. The initialization function, if assembled as shown in the preceding example, is completely compatible with C-language sources.

C.2.2 Create the Library Functions

You may create as many library functions as you like. A library function can have any number of parameters and any return type, but must be declared with the **PASCAL** and **FAR** attributes. The **PASCAL** attribute keeps the calling convention used by the library function consistent with the convention used by Windows. The **FAR** attribute permits the function to be called by applications.

The following example shows the general form of a library function:

```
HANDLE FAR PASCAL CreateInfo(lpInfo, nBytes)
LPSTR lpInfo;
int nBytes;
```

```
{
    HANDLE hInfo;
    PSTR pInfo;

    hInfo = LocalAlloc(LMEM_MOVEABLE | LMEM_ZEROINIT, nBytes);
    if (hInfo == NULL)
        return (-1);
    pInfo = LocalLock(hInfo);
    _lstrncpy(pInfo, lpInfo, nBytes);
    LocalUnlock(hInfo);
    return (hInfo);
}
```

Library functions should not use global or static variables to retain data that will be used in a subsequent call. Since Windows is a multitasking system, any other application could call a library function and modify a variable between the time it is set and the time it is needed. If you must save data between calls, either have the application pass a long pointer to a buffer that will hold the data, or allocate space in the library's local heap and return the local memory handle to the application. If a handle is returned, the application must supply it when requesting access to the data again.

A library function can call other functions within the library and even call functions in other Windows libraries. If you call functions in other libraries, you must import those functions by using one of the methods described in section C.5, "Linking with Functions in a Library."

A library function can call standard C run-time functions as long as they obey the **SS != DS** rule. In others words, library functions can call only those C run-time functions that do not assume that the data segment and stack are located in the same physical segment in memory. For a list of the C run-time functions and their segment assumptions, see *Microsoft Windows Programming Tools.*

If an application supplies the address of a callback function, the library function should assume that the address is a proper procedure-instance address. It is the application's responsibility to ensure that a procedure-instance address is created and passed to the function. A callback function can be called by using the following method:

```
int FAR PASCAL LibFunc(lpCallBackFunc)
FARPROC lpCallBackFunc;
{

    *(lpCallBackFunc)(1, 2, 3);
}
```

Not all functions in a library need to to be exported. If a library has helper functions that are called only by the exported functions, never from an application, the helper functions do not need to be exported.

C.2.3 Create the Module-Definition File

A Windows library's module-definition file must use the **LIBRARY** statement to define the library, and each function in the library must be explicitly exported by naming it in an **EXPORTS** statement. For example, a library named "info" that exports three functions should have the following statements:

```
LIBRARY info

DATA SINGLE

EXPORTS
    CreateInfo
    GetInfo
    DeleteInfo
```

The **DATA** statement is given to specify that the library has a data segment, but that only one copy of the segment will be created. You cannot have multiple instances of a library, which is contrary to conventions for Windows applications.

For convenience, you can also supply an explicit ordinal value for each exported function. An ordinal value is an integer that identifies the function and permits it to be imported by number instead of name. Each function receives an implicit ordinal number if you do not supply a number. In the example just cited, the ordinal values are 1, 2, and 3, respectively. You can override the implicit ordinal values by supplying your own, as in the following example:

```
EXPORTS
    CreateInfo @101
    GetInfo    @102
    DeleteInfo @103
```

C.2.4 Compile and Link

You compile the library source files as you would application sources. The following example shows an appropriate C-compiler command line for a small-model Windows library:

```
cl -c -Asnw -Gsw -Os -Zp select.c
```

Note that stack probes are disabled. Windows libraries should not check for stack overflow or underflow since the information used to check these conditions is not accessible to the library.

You link a Windows library by using the **link4** command, just as you would a Windows application. However, you must link with the appropriate C run-time library. For small-model libraries, the appropriate C run-time library is named *swinlibc.lib*. Use this library instead of the standard

C run-time library, *slibc.lib*. If your library uses any Windows functions, link with the appropriate Windows library. For example, use *slibw.lib* for small-model libraries.

The following example shows an appropriate **link4** command line for linking a small-model Windows library named "info":

```
link4 info,,, swinlibc slibw, info
```

Note

You must specify one of the libraries *swinlibc.lib*, *mwinlibc.lib*, *cwinlibc.lib*, or *lwinlibc.lib* when you link a Windows library. These are the small-, medium-, compact-, or large-model C run-time libraries for Windows libraries, respectively. Use these special libraries in place of the standard libraries *slibc.lib*, *mlibc.lib*, *clibc.lib*, or *llibc.lib*.

C.2.5 Add Resources

You add resources to a library exactly as you would add resources to an application. Create a resource script file and the necessary cursor, icon, and bitmap files, then compile the resource script file and add it to the executable file by using the **rc** command. The following example shows resources being compiled and added to a library named "info":

```
rc info.rc info.exe
```

C.2.6 Create the Import Library

You create an import library for your Windows library by using the **implib** command. This command uses your library's module-definition file to create a *.lib* file that can be specified on the **link4** command line for applications that call the library functions. An import library contains information about the name and ordinal value of the library functions. The linker uses this information to create entries in the application's executable file that Windows can use to dynamically link the library functions when the application is loaded.

The following example shows the appropriate **implib** command line for creating an import library for a Windows library named "info":

```
implib info.lib info.def
```

C.2.7 Create the Include File

You create an include file for a library by supplying appropriate function, type, and structure definitions. The following example shows a C-language include file for a Windows library named "info":

```
HANDLE FAR PASCAL CreateInfo(LPSTR, int);
int FAR PASCAL GetInfo(HANDLE, LPSTR, int);
int FAR PASCAL DeleteInfo(HANDLE);
```

The include file should be included in any application sources that call the library functions.

C.3 The Library Data Segment

A Windows library, like a Windows application, has a data segment in which global and static data declared within the library are stored. When an application calls a library function, the current data segment is automatically switched to the library's data segment, making access to the variables in the application's data segment no longer possible. This means the application must pass whatever data is required by the function as parameters to the function, or pass a long pointer to the data.

If a library function uses the global or static variables in the library's data segment to store information for a specific application, it must not assume that the information remains unchanged between calls from that application. Since Windows is a multitasking system, any number of other applications can call the same function between the first and second times the original application called it. If a function needs to save information for an application, the application should pass a long pointer to a buffer that will receive the information, or the function should allocate space in the library's local heap.

Since Windows does not require a library to have a stack, if no global or static variables are declared, then no data segment will be created for the library. In such cases, the data segment is not switched when the application calls a library function. In some cases, functions in a library can be denied access to the library's data segment by specifying the **NODATA** option in the module-definition file for that function. For such functions, no data-segment switching occurs when the application calls the function. Such functions must not attempt to access global or static variables in the library's data segment.

C.4 The Library Stack

A Windows library, unlike a Windows application, has no stack. Instead, it uses the caller's stack for parameters and local variables. This means the caller's stack must have sufficient space to handle the library's needs, and the library must take special care not to make excessive use of the stack (for example, a library must avoid using large character arrays as local variables).

Stack probes, which are provided by the C compiler to check for stack overflow and underflow, cannot be used in a Windows library since the information required to check the stack is in the application's data segment and is not available to Windows library functions.

C.4.1 The Local Heap

A Windows library can have a local heap. The local heap is subject to the same rules of usage as the local heap in a Windows application. To use a local heap, however, the library must initialize it by using the **LocalInit** function in the library's initialization function.

A local heap gives a library a means of storing data that is specific to a given application without exposing that data to overwriting if another application uses the library. In general, if a library needs to save data specific to an application, the library should allocate space in its local heap, copy the data to the allocated space, and return the data's local handle to the application. When the application wants the data back, it can supply the handle. Every handle in the library's local heap is guaranteed to be unique, so there will be no conflict with other applications.

The following example shows how to initialize and use a local heap in a Windows library. In this example, the local heap is used to store a structure containing information about an application. The library has three functions: CreateInfo, GetInfo, and DeleteInfo. The CreateInfo function allocates and fills a data block in the library's local heap. The GetInfo function retrieves information from the local heap. The DeleteInfo function removes the data block from the local heap.

To create this library, you need to do the following:

1. Use the **HEAPSIZE** statement in the module-definition file to establish an initial local-heap size. For libraries, the default size for a local heap is zero.
2. Create an initialization function for the library and use the **LocalInit** function to initialize the local heap.
3. Create the functions that use the local heap.

The first step is to set the local-heap size in the module-definition file. You need to determine an appropriate size and use the **HEAPSIZE** statement to set it, as in the following example:

```
LIBRARY info

DATA SINGLE

HEAPSIZE 4096

EXPORTS
    CreateInfo @1
    GetInfo @2
    DeleteInfo @3
```

The library name is "info". Since there can be only one instance of a library, the data segment is given the **SINGLE** attribute. The **HEAPSIZE** statement specifies 4096 bytes as the initial size of the library's local heap.

The next step is to create the initialization function. To do this, insert a call to the **LocalInit** function in the library's initialization function. The initialization function should look like this:

```
EXTERNFP <LocalInit>

cProc   LoadLib,<FAR,PUBLIC,NODATA>,<si,di>
cBegin
        xor     ax,ax
        jcxz    LoadError   ; Fail if no heap
        push    ax          ; LocalInit(0, 0, cbHeap)
        push    ax          ; Uses library's data segment
        push    cx
        call    LocalInit   ;Returns nonzero on success
LoadError:
cEnd
```

In this example, the initialization function calls the **LocalInit** function, passing it the size of the local heap and a default data-segment address. The initialization function returns the return value from **LocalInit**, so if the heap could not be initialized, the load will be canceled.

The final step is to create the library's functions. Once the local heap is initialized, there are no special steps the functions need to take. They simply call the **LocalAlloc**, **LocalLock**, **LocalUnlock**, and other local memory-management functions as needed. For example:

```
HANDLE FAR PASCAL CreateInfo(lpInfo, nBytes)
LPSTR lpInfo;
int nBytes;
```

```
{
    HANDLE hInfo;
    PSTR pInfo;

    hInfo = LocalAlloc(LMEM_MOVEABLE | LMEM_ZEROINIT, nBytes);
    if (hInfo == NULL)
        return (-1);
    pInfo = LocalLock(hInfo);
    _lstrncpy(pInfo, lpInfo, nBytes);
    LocalUnlock(hInfo);
    return (hInfo);

}

int FAR PASCAL GetInfo(hInfo, lpBuffer, wOffset, nBytes)
HANDLE hInfo;
LPSTR lpBuffer;
WORD wOffset;
int nBytes;
{
    PSTR pInfo;

    pInfo = LocalLock(hInfo);
    if (pInfo == NULL)
        return (-1);
    _lstrncpy(lpBuffer, pInfo+wOffset, nBytes);
    LocalUnlock(hInfo);
    return (nBytes);
}

int FAR PASCAL DeleteInfo(hInfo)
HANDLE hInfo;
{
    return (LocalFree(hInfo));
}
```

C.4.2 Initializing the Local Heap for Libraries

If the library allocates memory from the local heap and therefore uses the **HEAPSIZE** statement in the module-definition file, the initialization function must call the **LocalInit** function to initialize the local heap. For example, the following call initializes the heap for subsequent use:

```
LocalInit(0, 0, HeapSize);
```

The heap-size value in the **CX** register can be used in the **LocalInit** function call.

C.5 Linking with Functions in a Library

There are three ways an application can link with and use Windows library functions:

- Implicit import
- Explicit import
- **GetProcAddress** import

In each case, the application imports the library functions to be called. Implicit import is the preferred method since it requires no special treatment in an application's source files.

In an implicit import, the application contains calls to the library functions just as if the function were declared within the application. When the application is linked, the import library for the corresponding Windows library must be supplied on the linker command line. The import library contains the necessary information about the function in order to allow Windows to establish a dynamic link with the library when the application is loaded. This is the method used to link with such Windows functions as **CreateWindow** and **GlobalAlloc**. The Windows import library *slibw.lib* must be specified on the linker command line when you use these functions in your applications. In the following example, an application uses three functions, from the Windows library *info.exe*. The corresponding import library *info.lib* is given on the linker command line when the application is linked:

```
link4 app, , , slibw.lib info.lib, app
```

You can create an import library for each Windows library you create by using the **implib** command. This command converts information about exported functions found in your module-definition files into appropriate import-library entries.

In an explicit import, an application contains calls to the library functions as if the function were declared within the application. When the application is linked, its module-definition file must contain explicit **IMPORTS** statements that define the imported functions and the name of the library in which they are located. The linker uses this information to create entries that Windows can use to establish a dynamic link with the library when the application is loaded. In the following example, three functions, CreateInfo, GetInfo, and DeleteInfo, from the Windows library *info.exe* are explicitly imported:

```
IMPORTS
    Info.CreateInfo
    Info.GetInfo
    Info.DeleteInfo
```

Note that the module name of the Windows library is used, not the executable file name. The module name is the name given in the **LIBRARY** statement of that library's module-definition file.

In a **GetProcAddress** import, the application calls the functions indirectly and only after explicitly loading the library and dynamically linking to the desired function. Import libraries and/or **IMPORTS** statements are not required. In the following example, an application links dynamically with the CreateInfo function in the Windows library *info.exe*:

```
HANDLE hLibrary;
FARPROC lpFunc;

hLibrary = LoadLibrary("INFO.EXE");
if (hLibrary != NULL) {
    lpFunc = GetProcAddress(hLibrary, "CREATEINFO");
    if (lpFunc != (FARPROC) NULL)
        *(lpFunc)((LPSTR) Buffer, 512);
}
FreeLibrary(hLibrary);
```

In this example, the **LoadLibrary** function loads the desired Windows library and returns a module handle to the library. The **GetProcAddress** function retrieves the address of the CreateInfo function by using the function's name, "CREATEINFO". The name is spelled with capital letters since that is the way it is recorded in the library's executable file. The function address can then be used to call the function. The following statement is an indirect function call that passes two arguments (Buffer and the integer 512) to the function:

```
*(lpFunc)((LPSTR) Buffer, 512);
```

Finally, the **FreeLibrary** function frees the Windows library (removes it from memory) after it has been used.

C.6 A Sample Library: Select

This sample library contains functions that you can use to carry out selections by using the mouse. The functions are based on the graphics selection method described in Chapter 7, "The Cursor, the Mouse, and the Keyboard." These functions provide two kinds of selection feedback: a box that shows the outline of the selection, and a block that shows the entire selection inverted. The library exports the following functions:

Function	Action
StartSelection	Starts the selection and initializes the selection rectangle. When selecting with the mouse, you call this function when you receive a WM_LBUTTONDOWN message.
UpdateSelection	Updates the selection box or block. When selecting with the mouse, you call this function when you receive a WM_MOUSEMOVE message.
EndSelection	Ends the selection and fills in the selection rectangle with the final selection dimensions. When selecting with the mouse, you call this function when you receive a WM_LBUTTONUP message.
ClearSelection	Clears the selection box or block from the screen and empties the selection rectangle.

The selection rectangle is a **RECT** structure that the application supplies and the library functions fill in. The coordinates given in the rectangle are client coordinates.

To create this library you need to create several files:

File	Contents
select.c	The C-language source for selection functions
select.def	The module-definition file for the Select library
select.h	The include file for the Select library
select	The **make** file for the Select library
select.lib	The import library for the Select library

The Select library does not have an initialization file because the functions do not use a local heap and because no other initialization is necessary.

C.6.1 Create the Functions

You can create the library functions by following the description given in Chapter 7, "The Cursor, the Mouse, and the Keyboard." Simply copy the statements used to make the graphics selection into the corresponding functions. Also, to make the selection functions more flexible, add the additional block capability.

After you change it, the StartSelection function should look like this:

```
int FAR PASCAL StartSelection(hWnd, ptCurrent, lpSelectRect, fFlags)
HWND hWnd;
POINT ptCurrent;
LPRECT lpSelectRect;
int fFlags;
{
    if (!IsEmptyRect(lpSelectRect))
        ClearSelection(lpSelectRect);
    if (!fFlags & SL_EXTEND) {
        lpSelectRect->left = ptCurrent.x;
        lpSelectRect->top = ptCurrent.y;
    }
    lpSelectRect->right = ptCurrent.x;
    lpSelectRect->bottom = ptCurrent.y;
    SetCapture(hWnd);
}
```

This function receives a window handle, hWnd; the current mouse location, ptCurrent; a long pointer to the selection rectangle, lpSelectRect; and the selection flags, fFlags. The first step is to clear the selection if the selection rectangle is not empty. The **IsRectEmpty** function returns TRUE if the rectangle is empty. The StartSelection function clears the selection by calling the ClearSelection function, which is also in this library.

The next step is to initialize the selection rectangle. The StartSelection function extends the selection (it leaves the upper-left corner of the selection unchanged), if the SS_EXTEND bit in the fFlags argument is set. Otherwise, it sets the upper-left and lower-right corners of the selection rectangle to the current mouse location. The **SetCapture** function directs all subsequent mouse input to the window even if the cursor moves outside of the window. This is to ensure that the selection process continues uninterrupted. To call this function, an application would use the following statements:

```
case WM_LBUTTONDOWN:
    bTrack = TRUE;
    StartSelection(hWnd, MAKEPOINT(lParam), &SelectRect,
        (wParam & MK_SHIFT) ? SL_EXTEND : NULL);
    break;
```

After you change it, the UpdateSelection function should look like this:

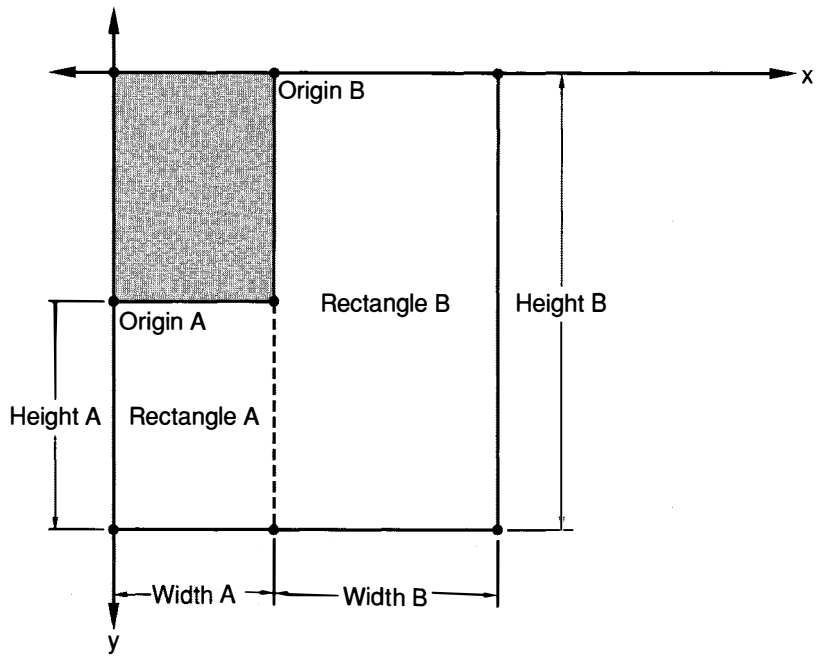
```
int FAR PASCAL UpdateSelection(hWnd, ptCurrent, lpSelectRect, fFlags)
HWND hWnd;
POINT ptCurrent;
LPRECT lpSelectRect;
int fFlags;
```

```
{
    HDC hDC;
    short OldROP;

    hDC = GetDC(hWnd);
    switch (fFlags & SL_TYPE) {
        case SL_BOX:
            OldROP = SetROP2(hDC, R2_XORPEN);
            MoveTo(hDC, lpSelectRect->left,
                   lpSelectRect->top);
            LineTo(hDC, lpSelectRect->right,
                   lpSelectRect->top);
            LineTo(hDC, lpSelectRect->right,
                   lpSelectRect->bottom);
            LineTo(hDC, lpSelectRect->left,
                   lpSelectRect->bottom);
            LineTo(hDC, lpSelectRect->left,
                   lpSelectRect->top);
            LineTo(hDC, ptCurrent.x, lpSelectRect->top);
            LineTo(hDC, ptCurrent.x, ptCurrent.y);
            LineTo(hDC, lpSelectRect->left, ptCurrent.y);
            LineTo(hDC, lpSelectRect->left, lpSelectRect->top);
            SetROP2(hDC, OldROP);
        break;

        case SL_BLOCK:
            PatBlt(hDC,
                lpSelectRect->left, lpSelectRect->bottom,
                lpSelectRect->right - lpSelectRect->left,
                ptCurrent.y - lpSelectRect->bottom,
                DSTINVERT);
            PatBlt(hDC, PrevX, OrgY,
                lpSelectRect->right, lpSelectRect->top,
                ptCurrent.x - lpSelectRect->right,
                ptCurrent.y - lpSelectRect->top, DSTINVERT);
        break;
    }
    lpSelectRect->right = ptCurrent.x;
    lpSelectRect->bottom = ptCurrent.y;
    ReleaseDC(hWnd, hDC);
}
```

As the user makes the selection, the UpdateSelection function provides feedback about the user's progress. For the box selection, the function first clears the current box by drawing over it, then draws the new box. This requires eight calls to the **LineTo** function.

To update a block selection, the UpdateSelection function inverts the rectangle by using the **PatBlt** function. To avoid flicker while the user selects, UpdateSelection inverts only the portions of the rectangle that are different from the previous selection rectangle. This means the function inverts two separate pieces of the screen. It assumes that the only area that needs inverting is the area between the previous mouse location and

the current. Figure C.1 shows the typical coordinates for describing the areas being inverted:



Rectangle A:

Origin A = (lpSelectRect->left, lpSelectRect->bottom)
Width A = (lpSelectRect->right - lpSelectRect->left
Height A = ptCurrent.y - lpSelectRect >bottom

Rectangle B:

Origin B = (lpSelectRect->right, lpSelectRect->top)
Width B = ptCurrent.x - lpSelectRect->right
Height B = ptCurrent.y - lpSelectRect->top

Figure C.1 Inverting a Rectangle

The first **PatBlt** call inverts the left-most rectangle by using lpSelectRect->left, the original location on the *x*-coordinate of the mouse button when first pressed, and lpSelectRect->bottom, the most recent update of the location of the mouse on the *y*-coordinate, to set the origin of the area to be inverted. The width of the first area is determined by

subtracting lpSelectRect->left from lpSelectRect->right, the most recent update of the location of the mouse on the *x*-coordinate. The height of this area is determined by subtracting lpSelectRect->bottom from ptCurrent.y, the current location of the mouse on the *y*-coordinate.

The second **PatBlt** call inverts the right-most rectangle by using lpSelectRect->right, the most recent location on the *x*-coordinate of the mouse button, and lpSelectRect->top, the original location on the *y*-coordinate of the mouse, to set the origin of the area to be inverted. The width of this second area is determined by subtracting lpSelectRect->bottom, the most recent update of the location of the mouse on the *x*-coordinate, from ptCurrent.x, the current location of the mouse on the *x*-coordinate. The height of this area is determined by subtracting lpSelectRect->top from ptCurrent.y, the current location of the mouse on the *y*-coordinate.

When the selection updating is complete, the values lpSelectRect->right and lpSelectRect->bottom are updated by assigning them the current values contained in ptCurrent.

To update a box selection, the application should call the UpdateSelection function as follows:

```
case WM_MOUSEMOVE:
    if (bTrack)
        UpdateSelection(hWnd, MAKEPOINT(lParam), &SelectRect, SL_BOX);
    break;
```

After you change it, the EndSelection function should look like this:

```
int FAR PASCAL EndSelection(ptCurrent, lpSelectRect)
POINT ptCurrent;
LPRECT lpSelectRect;
{
    if (ptCurrent.x < lpSelectRect->left) {
        lpSelectRect->right = lpSelectRect->left;
        lpSelectRect->left = ptCurrent.x;
    }
    else
        lpSelectRect->right = ptCurrent.x;
    if (ptCurrent.y < lpSelectRect->top) {
        lpSelectRect->bottom = lpSelectRect->top;
        lpSelectRect->top = ptCurrent.y;
    }
    else
        lpSelectRect->bottom = ptCurrent.y;
    ReleaseCapture();
}
```

The EndSelection function saves the current mouse position in the selection rectangle. For convenience, the final mouse position is checked to make sure it represents a point to the lower right of the original point. Rectangles typically are described by upper-left and lower-right corners.

If the final position is not to the lower right (that is, if either the *x*- or *y*-coordinate of the position is less than the original *x*- and *y*-coordinates), the values of the original point and the final point are swapped as necessary. The **ReleaseCapture** function is required since a corresponding **SetCapture** function was called. In general, you should release the mouse immediately after mouse capture is no longer needed.

Finally, when the user releases the left button, the application should call the EndSelection function to save the final point:

```
case WM_LBUTTONUP:
    bTrack = FALSE;
    EndSelection(MAKEPOINT(lParam), &SelectRect);
    break;
```

After you change it, the ClearSelection function should look like this:

```
int FAR PASCAL ClearSelection(hWnd, lpSelectRect, fFlags)
HWND hWnd;
LPRECT lpSelectRect;
int fFlags;
{
    HDC hDC;
    short OldROP;

    hDC = GetDC(hWnd);
    switch (fFlags & SL_TYPE) {
        case SL_BOX:
            OldROP = SetROP2(hDC, R2_XORPEN);
            MoveTo(hDC, lpSelectRect->left, lpSelectRect->top);
            LineTo(hDC, lpSelectRect->right, lpSelectRect->top);
            LineTo(hDC, lpSelectRect->right, lpSelectRect->bottom);
            LineTo(hDC, lpSelectRect->left, lpSelectRect->bottom);
            LineTo(hDC, lpSelectRect->left, lpSelectRect->top);
            SetROP2(hDC, OldROP);
        break;

        case SL_BLOCK:
            PatBlt(hDC,
                lpSelectRect->left, lpSelectRect->top,
                lpSelectRect->right - lpSelectRect->left,
                lpSelectRect->bottom - lpSelectRect->top,
                DSTINVERT);
        break;
    }
    ReleaseDC(hWnd, hDC);
}
```

Clearing a box selection means removing it from the screen. You can remove the outline by drawing over it with the XOR pen. Clearing a block selection means restoring the inverted screen to its previous state. You can restore the inverted screen by inverting the entire selection.

C.6.2 Create the Module-Definition File

To link the Select library, you need to create a module-definition file containing the following:

```
LIBRARY Select

CODE MOVEABLE
DATA NONE
HEAPSIZE 0

EXPORTS
    StartSelection
    UpdateSelection
    EndSelection
    ClearSelection
```

Since the selection functions do not use global or static variables and there is no local heap, the **DATA** statement is used to specify no data segment, and **HEAPSIZE** is used to set the heap size to zero.

C.6.3 Create the Include File

You need to create the *select.h* include file for the Select library. This file contains the definitions for the constants used in the functions, as well as function definitions. The include file should look like this:

```
int FAR PASCAL StartSelection(HWND, POINT, LPRECT, int);
int FAR PASCAL UpdateSelection(HWND, POINT, LPRECT, int);
int FAR PASCAL EndSelection(POINT, LPRECT);
int FAR PASCAL ClearSelection(HWND, LPRECT, int);
```

You should also use the include file in applications that use the selection functions. This will ensure that proper parameter and return types are used with the functions.

C.6.4 Compile and Link

To compile and link the Select library you need to create the **make** file as follows:

```
select.obj: select.c select.h
        cl -c -Asnw -Gsw -Os -Zp select.c

select.exe: select.obj
        link4 select, select.exe, , swinlibc slibw/NOD, select.def

select.lib: select.def
        implib select.lib select.def
```

Once you have compiled and linked the Select library, you can create a small test application to confirm that it is working properly. For a description of an application that uses the selection functions, see Chapter 13, "The Clipboard," or Chapter 9, "Bitmaps."