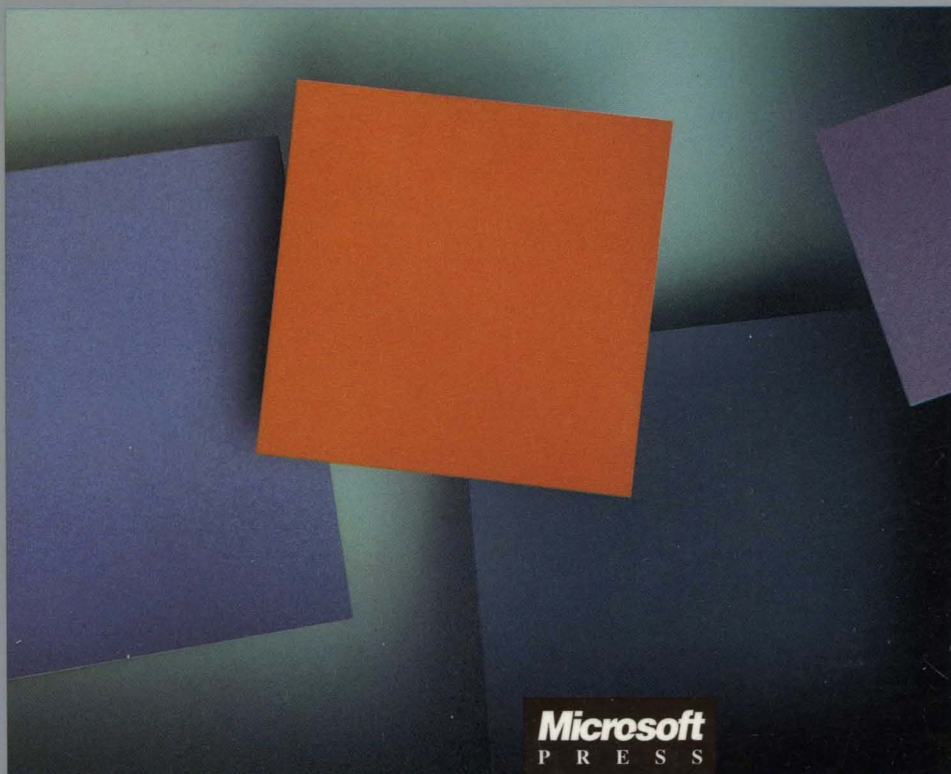
Microsoft® Programming Series

Microsoft®

WINDOWS™ 3

Developer's Workshop

- Debugging
- Dynamic Link Libraries
- Dynamic Data Exchange
- Scores of Source-Code Examples



Richard Wilton

Microsoft®

WINDOWS™ 3

Developer's Workshop

Microsoft® WINDOWS™ 3 Developer's Workshop

Richard Wilton

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 1991 by Richard Wilton

All rights reserved. No part of the contents of this book
may be reproduced or transmitted in any form or by any means
without the written permission of the publisher.

Library of Congress Cataloging-in-Publication Data

Wilton, Richard, 1953–
Microsoft Windows 3 developer's workshop / Richard Wilton.
p. cm. -- (Microsoft programming series)
Includes index.
ISBN 1-55615-244-2 (softcover) : $27.95 ($36.95 Can.)
1. Microsoft Windows (Computer programs) I. Title. II. Series.
QA76.76.W56W5 1991
005.4'3--dc20 91-29816
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 MLML 6 5 4 3 2 1

Distributed to the book trade in Canada by Macmillan of Canada, a division of
Canada Publishing Corporation.

Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging-in-Publication Data available.

Apple® and Macintosh® are registered trademarks of Apple Computer, Inc. Intel® is a registered trademark of Intel Corporation. AT®, IBM®, and PS/2® are registered trademarks of International Business Machines Corporation. Lotus® is a registered trademark of Lotus Development Corporation. CodeView®, Microsoft®, and MS-DOS® are registered trademarks and Windows™ is a trademark of Microsoft Corporation. OS/2® is a registered trademark licensed to Microsoft Corporation. Novell® is a registered trademark of Novell, Inc. Actor® is a registered trademark of The Whitewater Group, Inc.

Acquisitions Editor: Dean Holmes
Project Editor: Ron Lamb
Technical Editor: Mary DeJong

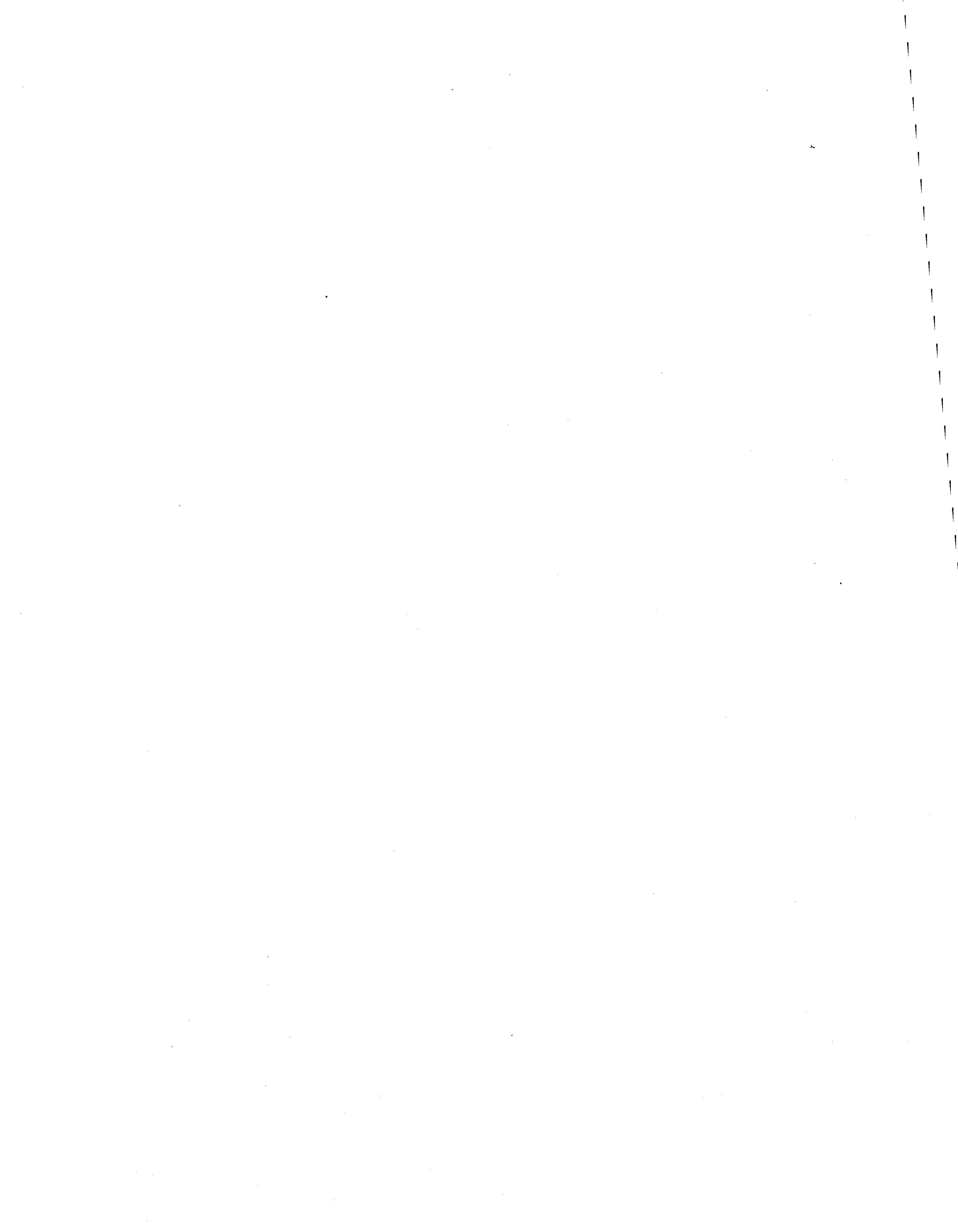
Contents

	Introduction	**ix**
Chapter 1:	**Windows Design**	**1**
Chapter 2:	**Debugging**	**27**
Chapter 3:	**Dynamic Link Libraries (DLLs)**	**59**
Chapter 4:	**Custom Controls**	**83**
Chapter 5:	**An Object-Oriented View**	**133**
Chapter 6:	**Dynamic Data Exchange (DDE)**	**157**
Chapter 7:	**Problems and Solutions**	**205**
Appendix:	**A Windows Programming Glossary**	**255**
	Index	**261**

Acknowledgments

This book came to life through the encouragement and support of my family, friends, colleagues, and students:

- Mom, Dad, and Mary Ellen, who patiently gave me the time and support I needed to concentrate on this book.
- Mike McCoy, Stan Stead, Kirk Andrews, and Bob Syarto, my very talented colleagues at UCLA, whose wide experience in object-oriented environments, networks, and multimedia applications always stimulates my interest and broadens my thinking about Windows programming; and Shawn Hall, a UCLA computer-science student and a skillful programmer who helped keep this book focused on how the Windows environment is designed.
- Mark Walsen, Tim Brewer, and David Weise of the Windows systems team and Don Hasson, Kraig Brockschmidt, and Bryan Woodruff of Microsoft product support for their thoughtful criticism and comments. I am especially grateful to Sanford Staab of the Windows systems team, who answered my many questions about DDEML programming with patience and clarity.
- And of course, the Microsoft Press team: Dave "Wild Man" Rygmyr, Dean Holmes, Dan Lipke, Ron Lamb, Mary DeJong, Cynthia Riskin, Judith Bloch, Debbie Kem, Patrick Forgette, Jennifer Harris, Ruth Pettis, Peggy Herman, Lisa Sandburg, Kim Eggleston, Rodney Cook, Susan Sherman, Alice Copp Smith, Katherine Erickson, and many others whose hard work helped this book evolve into the finished product you're now reading. Any errors in the text that have survived their careful scrutiny are certainly my own responsibility.



Introduction

This is a programmer's book about Windows programming. This book is not a tutorial, nor is it a compendium of tricks and wizardry. Instead, it is a collection of variations on a basic theme: The way to design Windows applications is to understand the design of the Windows environment itself.

My own experience with Windows programming grew out of my previous work with computer-video subsystems. In the mid-1980s, I spent several years developing application interfaces to IBM PC and PS/2 video systems, but I began to change my approach to application interface development when I started working with the Apple Macintosh and with Microsoft Windows.

I was initially not impressed with the Windows environment. In version 1 (1985), Windows' visual interface had a clunky, unconvincing appearance (Figure I-1). Worse, application development was an unpleasant experience without a robust debugger or insightful documentation. With version 2 (1987), Windows' appearance improved (Figure I-2), as did the development and debugging tools

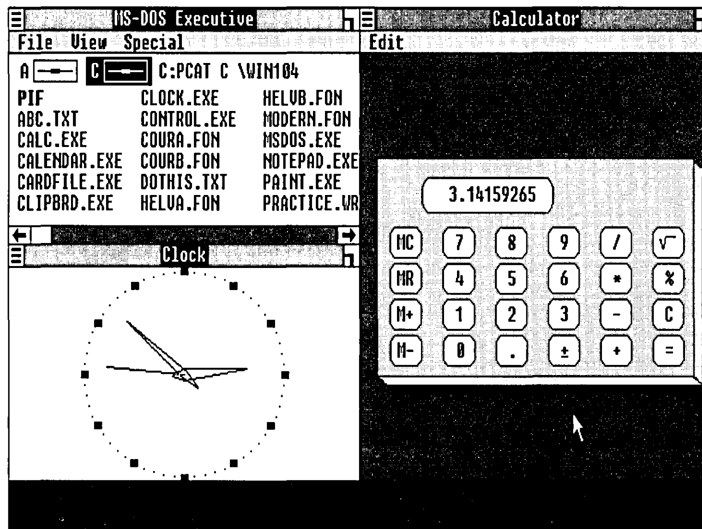


Figure I-1.
The user interface of Windows version 1, Microsoft's first graphical interface for IBM-compatible microcomputers.

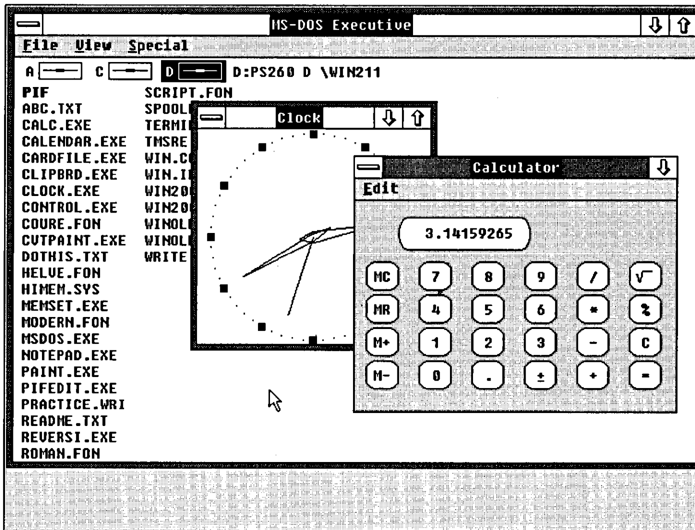


Figure I-2.
The user interface of Windows version 2, which introduced overlapping windows and support for high-resolution video displays.

in the Windows Software Development Kit. Probably the best thing about this version of Windows was the publication of Charles Petzold's encyclopedic *Programming Windows* book (Microsoft Press, 1990), which does a great job of describing the fundamentals of programming in the Windows environment.

Windows finally began to reach its potential with version 3 (1990), which introduced a new and further-improved visual appearance (Figure I-3). This version of Windows also solved some of the memory-management limitations of previous versions by running in protected mode on Intel 80286 and 80386 microprocessors. Most important, Windows 3 seemed to capture the imagination of application programmers. Many more commercial and public-domain applications have been written for Windows version 3 than for both of the previous versions.

As more people become involved in writing Windows programs, I think it makes sense to take a close look at the Windows environment from a software-design point of view. My goal in this book is to give you the ideas and the details you need to feel comfortable designing Windows applications.

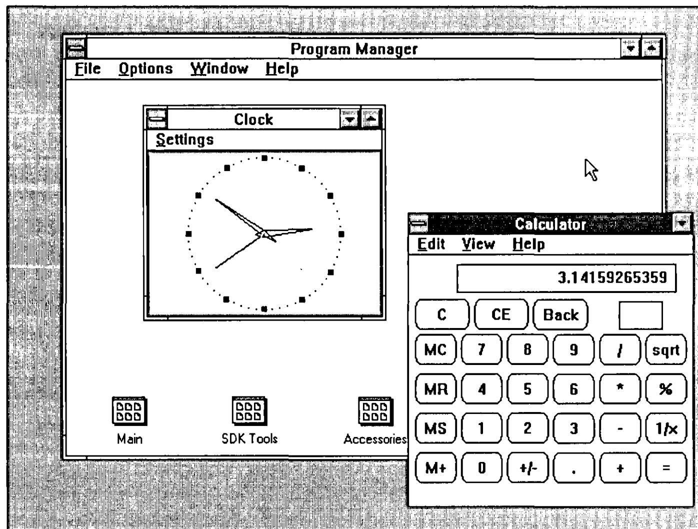


Figure I-3.
The user interface of Windows version 3, which uses proportionally spaced fonts and three-dimensional buttons and icons.

Using This Book

This is not a book for beginners. I assume that you know how to write a computer program, that you don't mind using the C programming language, and that you are somewhat familiar with the Windows programming environment. If you've never written a Windows application in C, you can use this book to learn about the design and structure of Windows programs, but you will find the source-code examples more useful if you already have some experience in designing Windows programs.

I don't intend this book to replace Microsoft's Windows Software Development Kit (SDK). The documentation supplied with the SDK is professionally written and thorough. You will certainly want to refer to it as you read about Windows and develop your own Windows applications.

It goes without saying that this book will be most useful if you have a set of robust software-development tools for Windows. Equally important, you need the documentation that describes the details of the Windows programming environment. As I wrote this book, I frequently referred to the manuals in the SDK and to my copy of Petzold's *Programming Windows.* You can't program in Windows unless you have good programming tools and thorough documentation.

You should also have a computer system that is powerful enough to let you write, run, and debug Windows programs. A minimal Windows development system is an 80286-based machine with a hard disk, at least 2 MB of memory, a VGA-compatible video subsystem, and a mouse. I used such a system—an IBM PS/2 Model 60—to develop many of the source-code examples in this book. However, the faster your computer, the less time you'll spend waiting for it to compile and link your applications.

You also need a Windows-compatible C compiler and linker. Microsoft's C compiler is still the most-used language translator for Windows applications, although a number of software vendors have introduced Windows-compatible compilers for C and for other high-level languages. The source-code examples in this book were created with Microsoft C, so you may have to work around some source-level incompatibilities if you compile the examples with another vendor's compiler.

In addition to a suitable computer system and a compiler, you should also have a set of Windows software-development tools, including a debugger and a dialog editor. In writing this book, I used the tools provided in Microsoft's SDK. By the time you read this, however, a number of other software vendors will also have released useful Windows development products.

How This Book Is Organized

The first chapter of this book gives you a bird's-eye view of the Windows environment. It emphasizes the important structural and functional components of Windows and shows how an application's design is based on the need to interact effectively with the Windows environment.

Chapter 2 is an overview of debugging in Windows. This chapter is the result of dozens of close encounters with foul and insidious bugs. It describes a variety of obvious and not-so-obvious bugs that could be lurking in your programs and suggests how you can detect and correct them.

Chapters 3 and 4 explore the programming of dynamic link libraries (DLLs) in Windows. Chapter 3 reviews the structure of DLLs in Windows. It covers some of the details of memory management, parameter passing, and the use of resources in DLLs. Chapter 3 lays the groundwork for Chapter 4, which focuses on a natural application of DLLs—namely, support for custom-control classes.

Chapter 5 examines Windows' object-oriented heritage. The Windows environment was designed by foresighted programmers who recognized the value of object-oriented concepts in a graphical windowing environment. This chapter

describes some object-oriented features of the Windows environment and suggests how Windows programs can benefit from object-oriented design.

Chapter 6 digs into Dynamic Data Exchange (DDE), Windows' protocol for interprocess communication. This chapter starts with an introduction to the client-server transaction model used in DDE. It then describes the message-based DDE protocol with an emphasis on the design of DDE transactions. This discussion leads naturally into the heart of the chapter—the DDE Management Library (DDEML). The chapter ends by considering some of the software design issues common to DDE programming.

Chapter 7 is a collection of typical Windows programming problems and solutions. The chapter describes some advanced programming techniques that are not really part of Microsoft's SDK documentation. There are no tricks or secrets here—just reasonable approaches to some programming puzzles that many Windows programmers will face sooner or later.

Source-Code Notes

I used Microsoft's Windows Software Development Kit, version 3.0, to build the source-code examples in this book. All the source-code examples were compiled with Microsoft C 6.0 (aka Microsoft C Professional Development System). The source code was tested in real, standard, and 80386 enhanced modes on several different computers, including an IBM PC/AT and IBM PS/2 models 60 and 70.

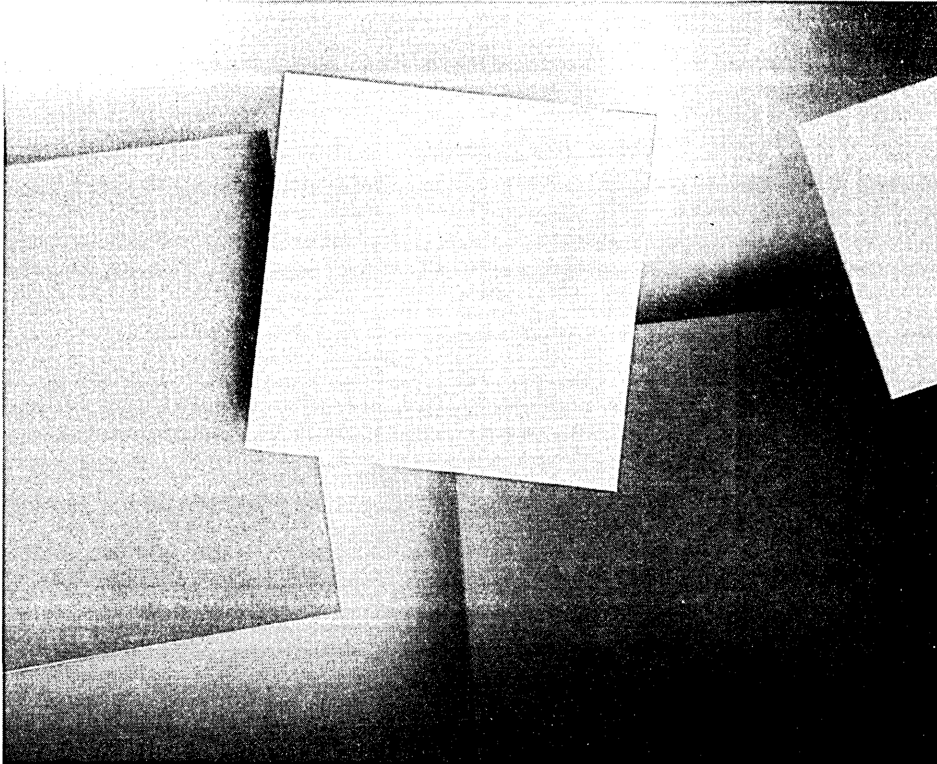
I have done my best to pare the source-code examples in this book to the minimum necessary to illustrate the techniques described in each chapter. None of the source code really represents a complete and polished Windows application. The most valuable part of a Windows application—the visual design, the flow of interactions with the user, the menus, and help text—can only be supplied by you.

If you see ways to improve on the techniques I've described, if you derive new applications from the source-code examples in this book, or if you spot bugs that need to be fixed, I'd like to hear about them. Please contact me at this address:

Microsoft Press
Attention: Windows: Developer's Workshop editor
One Microsoft Way
Redmond, WA 98052-6399

1

Windows Design



Although the Windows operating environment runs under MS-DOS, writing a Windows program is not the same as writing an MS-DOS program. It is unusual for a Windows program to call an MS-DOS function directly. Instead, Windows applications rely on the Windows application program interface (API) to obtain keyboard and mouse input, to produce output on a video display or printer, and to manage disk files.

In fact, the key to writing a successful Windows application lies in taking full advantage of Windows' built-in functionality. This may seem a daunting proposition in a programming environment that provides over 600 built-in functions. The way to grasp this wealth of built-in application support is to visualize how the Windows environment is put together.

The Structure of the Windows Environment

As you design a Windows program, you can rely on the Windows environment to provide three kinds of fundamental services. One is to perform basic input and output functions for the keyboard, mouse, video display, printers, disk files, and serial communications devices. Most of these functions are also supported to some extent in MS-DOS, but Windows' basic input/output (I/O) functions are much more comprehensive than those in MS-DOS. Windows' I/O functions are also designed to be device-independent. Hundreds of I/O devices are supported by Windows through a single set of API functions. Because of this power and generality, Windows programs almost always rely exclusively on the built-in API functions for input and output.

Another of Windows' fundamental jobs is to manage memory. Windows lets a program dynamically allocate and free blocks of memory. Windows' memory-management API gives programs transparent access both to expanded memory (bank-switched memory conforming to version 4.0 of the LIM EMS standard—the Lotus-Intel-Microsoft Expanded Memory Specification standard) and to extended memory (memory addressable above 1 MB). Windows also provides transparent support for virtual memory (sharing available memory by swapping blocks of memory to disk) in 80386-based and 80486-based computers.

Windows' third important service is to support multitasking—that is, to allow two or more programs to share the CPU, memory, and I/O hardware. The fact that Windows allows multitasking means that Windows' I/O and memory-management functions accommodate the need for different programs to share resources cooperatively.

You can imagine the Windows environment to be structured as a set of "managers" that are responsible for supporting I/O, memory management, and multitasking. In fact, Windows programmers refer to the "I/O manager," the "memory manager," the "task manager," and the "window manager" as if they were discrete components of Windows. Although this is a convenient conceptual model, these functions correspond only roughly to separate modules in the Windows environment.

Modules

In Windows, a module is any collection of executable code or data that can be loaded into memory. A module might contain a user-written application, a hardware device driver, or a dynamic link library (DLL) of functions or data resources that a program contained in another module can access. In Windows applications, as in the Windows environment itself, support for complex operations is often distributed across several Windows modules.

Windows itself is built from a number of interrelated modules. The Windows API functions are implemented in a set of modules that are loaded into memory when Windows starts up. These modules are shown in Figure 1-1. The three main modules are GDI.EXE; USER.EXE; and KERNEL.EXE, KRNL286.EXE, or KRNL386.EXE (depending on whether Windows is running in real, standard, or enhanced mode). These three modules contain most of the API functions. The remaining functions, which are used for accessing I/O hardware, are supported in a set of device-driver modules (COMM.DRV, KEYBOARD.DRV, and so on).

To write a Windows application, however, you rarely need to worry about the names of the modules that make up the Windows environment. Your programs can call any Windows API function regardless of which module contains the function. Usually, the only modules you need to work with explicitly are the ones that you write—that is, the modules that contain your application programs and DLLs.

Windows has the ability to load a module dynamically at the time an executing program needs to access the module's functions or data. Windows supports several API functions (*LoadLibrary*, *LoadModule*, and *WinExec*) that let a program explicitly load a module at the time the program executes. Windows can also load a module implicitly if another program refers to a function within the module. Such implicit dynamic binding between functions in different modules relies on the notion of exported and imported functions.

Module	***Functions Supported***
GDI.EXE	Graphics device interface: output of graphics images, color-palette management
USER.EXE	Window, icon, and cursor management
KERNEL.EXE (real mode) KRNL286.EXE (standard mode) KRNL386.EXE (enhanced mode)	Memory management, task scheduling
COMM.DRV	Device driver for serial communications
KEYBOARD.DRV	Device driver for keyboard
MOUSE.DRV	Device driver for mouse
SOUND.DRV	Device driver for sound generation
SYSTEM.DRV	Device drivers for system timer, disk drives

Figure 1-1.
Modules in the Windows environment. In addition to these main modules and device drivers, additional device drivers for video displays, networks, and other hardware can be installed.

Functions

Most Windows modules that contain executable code implement one or more functions that can be called by code in other modules. Such functions are known as exported functions. The module that contains the function's executable code exports the function; a module that calls the function imports the function. Exported functions are the only functions in a module that can be called by programs in other modules.

Exported functions are used extensively both within Windows itself and in Windows applications. The entire Windows API consists of functions that are exported from the various modules that make up the Windows environment. Moreover, Windows expects applications to define exported functions that Windows itself will call. In particular, Windows calls certain functions exported from an application to direct keyboard and mouse input to the application and to facilitate multitasking.

Tasks and Instances

Windows supports cooperative multitasking. Windows' task manager maintains a list of tasks and keeps track of the order in which the tasks execute. When

Windows transfers control to a task, other tasks cannot run until the task in control explicitly yields control to the task manager. Windows' cooperative multitasking scheme differs from a preemptive multitasking implementation such as the one used in OS/2. Under preemptive multitasking, the operating system executes each task in turn for a predetermined amount of time and switches from task to task regardless of whether the tasks themselves yield control to the operating system.

Windows executes different applications as different tasks. Also, when you run two or more instances of an application at once, Windows executes each instance of the application as a separate task. Windows loads a separate, unshared copy of the application's default data segment into memory for each instance. Task-specific data, including the stack and local data, are stored in the data segment that corresponds to each instance of an application.

Although Windows loads a copy of the module's default data segment for each instance of the module, there is only one copy of a module's executable code in memory regardless of how many instances are running. For this reason, executable code and data are strictly segregated in a Windows application. You must pay careful attention to this requirement if you are programming in assembly language, but separating code and data is not a concern if you use a high-level language translator such as the Microsoft C compiler to generate the executable code for your applications.

Furthermore, Windows' cooperative multitasking scheme imposes other constraints on application design. For cooperative multitasking to work, each Windows application must be designed with at least one window function—a function that can be called by Windows' task manager—and a message loop that yields control to the task manager. (See the next section for a discussion of messages.) This particular requirement is important because it imposes a certain logical structure on all Windows applications.

Figure 1-2 illustrates the flow of a Windows application that cooperates properly with the task manager. The application's central control structure is a loop in which the application carries out a time-limited action and then transfers control back to Windows' task manager. The task manager allows other tasks to execute. It then transfers control back to the application to carry out another action.

This structure is simple yet powerful. The key to its usefulness is that an application can carry out a different action each time it gains control from the task manager. Implicit in this design, however, is a mechanism for an application to determine which of a set of possible actions to carry out. In Windows, messages are the mechanism for doing this.

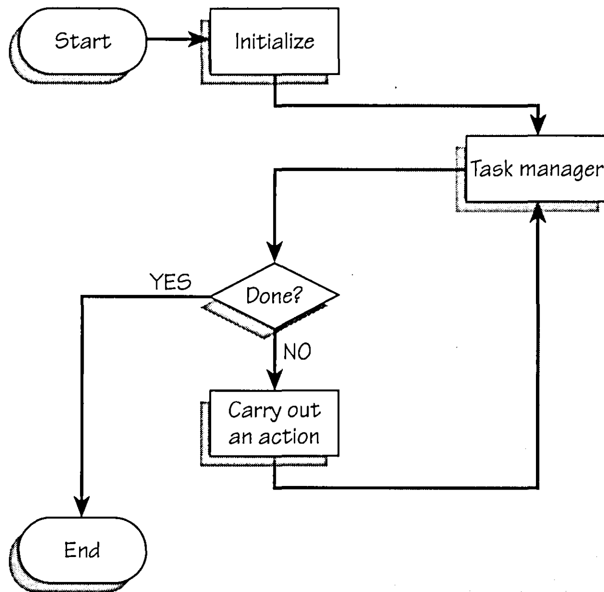


Figure 1-2.
Flow of control in a Windows application.

Messages

In the Windows environment, a message is a stereotyped set of data that is passed through a function call from Windows to an application's window function. The function call always uses the set of parameters that is shown in Figure 1-3 on the following page. The second parameter is a value that identifies the message. The Microsoft Windows Software Development Kit (SDK) associates symbolic names with message identifiers. Symbolic names such as WM_CREATE, WM_COMMAND, and WM_PAINT suggest how the messages they identify are to be interpreted by the application that receives them. (The Windows SDK documentation details all of the messages and their parameters.)

Windows' use of messages might be construed as a kind of event-notification mechanism. Different events in the Windows environment give rise to different messages. Some events are intuitively obvious. For example, when you click a mouse button or press a key on the keyboard, Windows uses one or more messages to indicate exactly which mouse or keyboard activity has occurred. That is, Windows calls an application's window function with message-identifier values such as WM_LBUTTONDOWN, WM_MOUSEMOVE, or WM_KEYDOWN. The

```
LONG PASCAL FAR
WndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
  /* hWnd:     Window handle
     wMsg:     Message identifier
     wParam:   Usage depends on the value of wMsg
     lParam:   Usage depends on the value of wMsg */
  :
}
```

Figure 1-3.
A C-language declaration of a message-processing function in a Windows application.

application responds to each message by processing the mouse-location or keystroke information in the associated parameters *wParam* and *lParam.*

Windows also uses certain messages to notify an application to take specific actions. For example, Windows uses the WM_PAINT message to notify an application to produce output on the video display. An application responds to this particular message by calling the Windows API functions that draw text or graphics on the screen. Windows uses other messages to notify an application about changes in the state of the Windows environment and to facilitate direct communication between different applications.

To pass a message to an application, Windows can either post the message to an application-specific queue or send the message directly to a window function in the application. For example, messages that notify an application of keyboard or mouse input are always posted to the message queue. To process messages properly, every application must contain a message loop that checks the queue periodically for the arrival of newly posted messages, as shown in Figure 1-4. The Windows API provides a set of functions (*GetMessage, PeekMessage, WaitMessage*) that do this efficiently.

In contrast, Windows sends messages such as window-management messages directly to an application, bypassing the application's message queue. The difference between the two message-passing methods is one of synchronization between Windows and the application. When a message is posted to an application's message queue, the application does not actually process it until Windows' task manager returns control to the application. When a message is sent directly to an application, the application processes it immediately.

```
int PASCAL WinMain( ... )
{
    MSG    msg;


    Initialize( ... );

    /* yield to the task manager; loop until done */
    while( GetMessage( &msg, ... ) )
    {
      TranslateMessage( &msg );          /* process the message */
      DispatchMessage( &msg );
    }

    return ... ;
}
```

Figure 1-4.
A typical message-processing loop in a Windows application. Compare the flow of control in this example with the flowchart in Figure 1-2.

Windows

In the Windows environment, the entity that processes messages in an application is a window. There are two logical components to a window, a data structure that describes the window's characteristics and a window function that processes messages. The data structure is created and maintained internally by Windows' window manager. The window function is an exported function that can be called directly by Windows.

Intuitively, the purpose of a window is to carry out the graphical processing required to maintain a visual window on the screen. However, you might better regard a Windows window as a message-processing unit that does not necessarily have a visual component. An application can create a window whose purpose is only to do a specific message-processing job without ever appearing on the display screen. For example, an application that uses the message-based Dynamic Data Exchange (DDE) protocol to exchange messages with other applications can create one or more "invisible" windows that encapsulate DDE message processing.

Much of the power of Windows programming is based on the fact that an application can create multiple windows, each with a different function. Windows' window manager facilitates this by maintaining a list of the windows created by all executing applications. The window manager associates windows with the tasks in which they are created. When a task terminates, the window manager destroys the windows that the task created if the task itself has not already destroyed them.

Multiple windows in a task are organized in a hierarchy of owners and owned windows—that is, each window can be owned by another window, and each window can have one or more owned windows associated with it. These owner/owned relationships affect both the lifetime of a window and the manner in which a window defined with particular styles is visually displayed, as shown in Figure 1-5. When a window has an owner, the window is destroyed implicitly if its owner is destroyed. Also, windows without owners can be overlapped by other windows, but owned windows are always displayed in front of their owners, as shown in Figure 1-6.

A special owner/owned relationship exists for windows created with the WS_CHILD style. For example, a child window can be displayed only within the rectangular area defined by its parent (owner). Also, Windows' window manager treats parent and child windows differently in regard to the kinds of messages they receive. In particular, Windows sends a notification message, WM_PARENTNOTIFY, to a parent window whenever a child window is created or destroyed. A child window uses a different message, WM_COMMAND, to notify a parent window that the child has processed user input.

Visual Style	***Is Owner Specified?***	***Lifetime***	***Visual Relationships***
WS_OVERLAPPED	No	Task	Overlaps other windows Always has caption and border *CreateWindow* can specify default location and size
WS_OVERLAPPED	Yes	Owner	Overlaps owner Always has caption and border Not visible if owner is not visible
WS_POPUP	No	Task	Overlaps other windows
WS_POPUP	Yes	Owner	Overlaps owner Not visible if owner is not visible
WS_CHILD	Yes	Owner (parent)	Clipped within parent

Figure 1-5.
Characteristics of windows with different visual styles and owners. A window's style and owner are specified at the time a window is created.

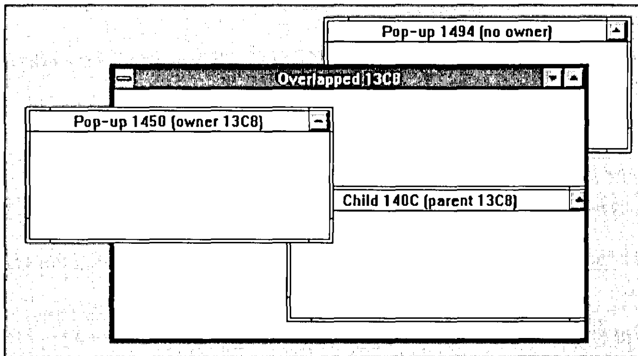


Figure 1-6.
An example of the visual relationships of windows with and without owners. The overlapped window (13C8) owns both a pop-up window (1450) and a child window (140C). The child window is clipped by the owner, and the owned pop-up window overlaps the owner, but the pop-up without an owner (1494) can be overlapped by the other windows in the task.

In the Windows environment, the processing necessary to keep track of window relationships is hidden from applications. Internally, the window manager maintains a list of data structures, each of which identifies and describes a window. From an application's point of view, however, each window is identified only by a unique integer value assigned by the window manager. This identifying value is the window's handle.

Handles

Windows are only one of a variety of items that are identified by handles in the Windows environment. The following items also use handles:

- Modules
- Tasks
- Instances
- Files
- Blocks of memory
- Menus
- Controls
- Fonts

- Resources (icons, cursors, strings, and so on)
- GDI objects (bitmaps, brushes, metafiles, palettes, pens, regions) and device contexts

Windows programs do not use physical addresses to identify blocks of memory, files, tasks, or dynamically loaded modules. Instead, the Windows API identifies these items by assigning handles to them and passing the handle values to an application.

A Windows application can obtain a handle for a particular item in one of two different ways. Many API functions, including *CreateWindow*, *GlobalAlloc*, and *OpenFile*, return a handle. Also, Windows can pass a handle to an application as a parameter in a call to an exported function in the application. As you might infer from the variety of items that are identified by handles, you can expect to work with handles just about everywhere in the Windows environment. This pervasive use of handles affects the design of every Windows program.

The value of a handle is meaningful only in that it uniquely identifies the item it represents. In general, handle values correspond to entries in a list of items, but only Windows itself can access the list directly. An application can usually access the item that a handle represents only by calling a Windows API function that specifically dereferences the handle. For example, an application can allocate a block of memory for its own use and obtain a handle to the block by calling the API function *GlobalAlloc*:

```
hMem = GlobalAlloc( ... );
```

The handle returned by *GlobalAlloc* identifies a particular block of memory to the application, but the application cannot use the handle to access the memory block directly. Instead, the application must call another API function, *GlobalLock*, which returns the physical address of the memory block:

```
lpMem = GlobalLock( hMem );
```

Memory Management

Windows' memory manager controls all available system memory. It dynamically allocates and frees blocks of memory as needed for code and data segments from modules. The memory manager also supports a set of API functions that a Windows program can call to allocate blocks of memory for its own use.

From the point of view of a Windows program, the memory manager organizes available memory in two different heaps: a global heap and a local heap. The global heap encompasses all available memory. The memory manager allocates global memory on a segment-by-segment basis. For example, if an application contains two executable-code segments and one default data segment, Windows loads the three segments into three separate blocks of global memory. Similarly, each call to the API function *GlobalAlloc* allocates a block of memory that an application can access by using a far pointer.

The local heap consists of memory in a module's default data segment, which is allocated from the global heap when the module is loaded into memory. A program can call *LocalAlloc* to allocate blocks of memory in the local heap. These memory blocks are addressable as near data. Because a module's local heap shares a single memory segment with the module's stack and static data variables, the amount of data that can be stored in the local heap is limited by the size of a physical segment in memory (64 KB) and the amount of memory used by the stack and static variables.

The essential difference between the local and global heaps is one of scale. The local heap is best for containing small amounts of data used only in a single instance of an application. Use the global heap for large memory blocks and for data shared among two or more modules.

The Structure of a Windows Application

All Windows applications bear a certain resemblance to each other. This is because the structure of a Windows application is mostly determined by the need for it to interact smoothly with the Windows environment. To build a Windows program, you use the structural components that characterize the Windows environment: modules, functions, tasks, instances, messages, windows, handles, and dynamically allocated memory.

You can see this in the structure of the sample application in Figure 1-7 on the following page. The actions the program carries out actually have little to do with the program's design. If a different programmer had written this program, you would notice differences in style, but the flow of control and the modular structure of the application would be pretty much the same.

```
#**********************************************************************
#                                                                     *
# NMAKE description for MODSTAT.EXE                                   *
#                                                                     *
#**********************************************************************

.c.obj:
    cl /AM /c /G2sw /Osw /W4 /Zlp $*.c


ALL:    modstat.exe

modstat.obj:    modstat.c modstat.h

modstat.res:    modstat.rc modstat.h modstat.ico
    rc /r modstat.rc

modstat.exe:    modstat.obj modstat.res modstat.def
    link /al:16 /nod /noe modstat, , , libw mlibcew, modstat.def
    rc modstat.res
```

```
/**********************************************************************
*                                                                     *
* MODSTAT.C                                                           *
*                                                                     *
* Exports:      TopLevelWndFn                                         *
*               WndEnumFn                                             *
*                                                                     *
**********************************************************************/

#define     NOCOMM
#include    <windows.h>
#include    <string.h>      /* contains strrchr and strchr */
#include    "modstat.h"


/*** FUNCTION PROTOTYPES ***/

LONG PASCAL FAR TopLevelWndFn( HWND, WORD, WORD, LONG );
BOOL PASCAL FAR WndEnumFn( HWND, DWORD );

static HWND     Init( HANDLE, HANDLE, int );
static void     MsgCommand( HWND, WORD, LONG );
```

Figure 1-7. *(continued)*
Source code for MODSTAT.EXE.

Figure 1-7. *continued*

```
static void     MsgPaint( HWND );
static void     ShowModuleInfo( HDC );


/*** GLOBAL VARIABLES ***/

char    szTopLevelClass[] = "ModStat:TopLevel";
char    szAppTitle[] = "System Modules";
HANDLE  hInstance = 0;
int     nCharX, nCharY;

struct
{
  int   CX;
  int   CY;
}
        CharSize;


/************************************************************************
*                                                                       *
* WinMain                                                               *
*                                                                       *
************************************************************************/

int PASCAL
WinMain( HANDLE hInst, HANDLE hPrevInst, LPSTR lpszCmdLine, int nCmdShow )
{
    HWND    hWnd;
    MSG     msg;


    hWnd = Init( hInst, hPrevInst, nCmdShow );
    if( !hWnd )
      return 0;

    while( GetMessage( &msg, 0, 0, 0 ) )
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }

    return msg.wParam;
}
```

(continued)

Figure 1-7. *continued*

```
/*********************************************************************
 *                                                                   *
 * Init                                                              *
 *                                                                   *
 *********************************************************************/

static HWND Init( HANDLE hInst, HANDLE hPrevInst, int nCmdShow )
{
    WNDCLASS    wc;
    HWND        hWnd;
    HDC         hDC;
    TEXTMETRIC  tm;


    if( hPrevInst )
    {
      /* copy global data from previous instance */
      GetInstanceData( hPrevInst, (NPSTR)&CharSize, sizeof CharSize );
    }

    else
    {
      /* register the top-level window class */
      wc.lpszClassName    = szTopLevelClass;
      wc.hInstance        = hInst;
      wc.lpfnWndProc      = TopLevelWndFn;
      wc.hCursor          = LoadCursor( 0, IDC_ARROW );
      wc.hIcon            = LoadIcon( hInst, "TopLevelIcon" );
      wc.lpszMenuName     = "TopLevelMenu";
      wc.hbrBackground    = COLOR_WINDOW+1;
      wc.style            = CS_HREDRAW | CS_VREDRAW;
      wc.cbClsExtra       = 0;
      wc.cbWndExtra       = 0;

      if( !RegisterClass( &wc ) )
        return 0;                        /* return 0 if unsuccessful */

      /* save the default character dimensions */
      hDC = CreateDC( "DISPLAY", NULL, NULL, NULL );
      GetTextMetrics( hDC, &tm );
      DeleteDC( hDC );

      CharSize.CX = tm.tmAveCharWidth;
      CharSize.CY = tm.tmHeight + tm.tmExternalLeading;
    }

    /* save the current instance handle in a global variable */
    hInstance = hInst;
```

(continued)

Figure 1-7. *continued*

```
    /* create and display a top-level window */
    hWnd = CreateWindow( szTopLevelClass,
                         szAppTitle,
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                         0,
                         0,
                         hInstance,
                         NULL );

    ShowWindow( hWnd, nCmdShow );
    UpdateWindow( hWnd );

    return hWnd;
}


/************************************************************************
*                                                                       *
* TopLevelWndFn                                                         *
*                                                                       *
************************************************************************/

LONG PASCAL FAR
TopLevelWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bDWP = FALSE;


    switch( wMsg )
    {
      case WM_PAINT:
        MsgPaint( hWnd );
        break;

      case WM_COMMAND:
        MsgCommand( hWnd, wParam, lParam );
        break;

      case WM_DESTROY:
        PostQuitMessage( 0 );
        break;

      default:
        bDWP = TRUE;
        break;
    }
```

(continued)

Figure 1-7. *continued*

```
    if( bDWP )
      lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

    return lRVal;
}


/*************************************************************************
*                                                                       *
* MsgCommand                                                            *
*                                                                       *
*************************************************************************/

static void MsgCommand( HWND hWnd, WORD wParam, LONG lParam )
{
    switch( wParam )
    {
      case IDM_ENUM:
        InvalidateRect( hWnd, NULL, TRUE );
        break;

      default:
        break;
    }
}


/*************************************************************************
*                                                                       *
* MsgPaint                                                              *
*                                                                       *
*************************************************************************/

static void MsgPaint( HWND hWnd )
{
    HDC         hDC;
    PAINTSTRUCT ps;


    InvalidateRect( hWnd, NULL, TRUE ); /* update the entire client area */

    hDC = BeginPaint( hWnd, &ps );
    ShowModuleInfo( hDC );
    EndPaint( hWnd, &ps );
}
```

(continued)

Figure 1-7. *continued*

```
/*********************************************************************
 *                                                                   *
 * ShowModuleInfo                                                    *
 *                                                                   *
 *********************************************************************/

static void ShowModuleInfo( HDC hDC )
{
    FARPROC     pEnumFn;
    char        szBuf[80];
    HANDLE      hModule;
    char        szModuleName[12];
    int         nRefCount;
    static char szOtherInst[] = "Other instances of this application";


    /* display task and instance information */
    wsprintf( szBuf, "This is one of %d currently executing tasks.",
                    GetNumTasks() );
    nCharX = CharSize.CX;
    nCharY = CharSize.CY;
    TextOut( hDC, nCharX, nCharY, szBuf, lstrlen(szBuf) );

    wsprintf( szBuf, "Task handle: %04X  Instance handle: %04X",
                    GetCurrentTask(), hInstance );
    nCharY += CharSize.CY;
    TextOut( hDC, nCharX, nCharY, szBuf, lstrlen(szBuf) );

    /* display module handle, name, and reference count */
    GetModuleFileName( hInstance, szBuf, sizeof szBuf );
    lstrcpy( szModuleName, strrchr( szBuf, '\\' )+1 );
    *(strchr( szModuleName, '.' )) = 0;
    hModule = GetModuleHandle( szModuleName );
    nRefCount = GetModuleUsage( hModule );
    wsprintf( szBuf, "Module handle: %04X  Name: %s  Reference count = %d",
                    hModule, (LPSTR)szModuleName, nRefCount );
    nCharY += CharSize.CY;
    TextOut( hDC, nCharX, nCharY, szBuf, lstrlen(szBuf) );

    if( nRefCount > 1 )
    {
      nCharY += 2 * CharSize.CY;
      TextOut( hDC, nCharX, nCharY, szOtherInst, (sizeof szOtherInst) - 1 );
```

(continued)

Figure 1-7. *continued*

```
        /* enumerate other instances of this application */
        pEnumFn = MakeProcInstance( (FARPROC)WndEnumFn, hInstance );
        EnumWindows( pEnumFn, MAKELONG(hModule, hDC) );
        FreeProcInstance( pEnumFn );
      }
}


/************************************************************************
 *                                                                      *
 * WndEnumFn                                                            *
 *                                                                      *
 ************************************************************************/

BOOL PASCAL FAR WndEnumFn( HWND hWnd, DWORD dwParam )
{
    char    szBuf[64];
    HANDLE  hInst;
    HANDLE  hModule;


    /* determine the module name for this window */
    hInst = GetWindowWord( hWnd, GWW_HINSTANCE );
    GetModuleFileName( hInst, szBuf, sizeof szBuf );
    lstrcpy( szBuf, strrchr( szBuf, '\\' )+1 );
    *(strchr( szBuf, '.' )) = 0;

    hModule = GetClassWord( hWnd, GCW_HMODULE );

    /* display task, module, and instance handle for other instances */
    if( (hModule == LOWORD(dwParam)) && (hInst != hInstance) )
    {
      wsprintf( szBuf, "Task handle: %04X  Instance handle: %04X",
                       GetWindowTask( hWnd ), hInst );
      nCharY += CharSize.CY;
      TextOut( HIWORD(dwParam), nCharX, nCharY, szBuf, lstrlen(szBuf) );
    }

    return TRUE;
}
```

(continued)

Figure 1-7. *continued*

```
/*********************************************************************
*                                                                    *
* MODSTAT.RC resource script                                         *
*                                                                    *
*********************************************************************/

#include "modstat.h"

/* icon */
TopLevelIcon    ICON modstat.ico

/* menus */
TopLevelMenu    MENU
{
  MENUITEM     "&Enumerate"     IDM_ENUM
}
```

```
/*********************************************************************
*                                                                    *
* MODSTAT.H                                                          *
*   Header file for MODSTAT.C                                        *
*                                                                    *
*********************************************************************/

#define IDM_ENUM        101
```

```
;*********************************************************************
;                                                                    *
; MODSTAT.DEF module-definition file                                 *
;                                                                    *
;*********************************************************************

NAME            MODSTAT
DESCRIPTION     'MODSTAT.EXE version 1.0'
EXETYPE         WINDOWS
STUB            'WINSTUB.EXE'

CODE            LOADONCALL MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE MULTIPLE
```

(continued)

Figure 1-7. *continued*

```
SEGMENTS        _TEXT   PRELOAD MOVEABLE DISCARDABLE

HEAPSIZE        1024
STACKSIZE       5120

EXPORTS         TopLevelWndFn
                WndEnumFn
```

The sample application displays information about the modules loaded in memory. The application's module name, MODSTAT, is specified in the NAME statement in the module-definition file, MODSTAT.DEF. The corresponding executable file is MODSTAT.EXE. The module exports two functions: *TopLevelWndFn* and *WndEnumFn*. Both of these functions are compiled as far functions that use the Pascal convention for parameter-passing, and both are listed as exports in the module-definition file. These functions are exported so that Windows itself can call them; the far and Pascal calling conventions are required for all exported functions that can be called by Windows.

Windows calls *TopLevelWndFn* to process messages for the application's top-level window (its only window), and *WndEnumFn* in response to the application's call to *EnumWindows*. A single call to *EnumWindows* causes Windows to call *WndEnumFn* once for each overlapped and pop-up window in the window manager's list.

If you run multiple instances of this application, you can see in each instance's display the relationship between modules, tasks, and instances. Each time you invoke MODSTAT.EXE, Windows creates a new task and a new instance and assigns handles to each. All instances of the application, however, are associated with the same module and module handle.

Initialization

When you invoke MODSTAT.EXE, Windows loads the executable code and default data segment into memory. It then transfers control to a short initialization function (named *__astart* in programs compiled with the Microsoft C compiler) that initializes the application's stack and local heap in the default data segment and then transfers control to *WinMain*.

WinMain

Just as *main* is the function where a standard C program begins executing, *WinMain* is the function where the execution of a Windows application begins. Usually, *WinMain*'s first action is to perform the initialization required to support an instance of the application. In the example in Figure 1-7, *WinMain* calls a function named *Init* to do this initialization. *Init*'s most important actions are embodied in its calls to *RegisterClass* and *CreateWindow*. The call to *RegisterClass* describes the default characteristics of the top-level window in the application. *Init* calls *RegisterClass* only for the first instance of the application. After the class has been registered, Windows maintains the class description internally so that subsequent instances of the application can refer to it. Windows discards the class description only when all instances of the application have terminated.

From the point of view of program flow of control, the crucial data element in the call to *RegisterClass* is the address of *TopLevelWndFn*, which is assigned to *wc.lpfnWndProc*. After *Init* calls *CreateWindow* to create a window using the registered class description, Windows can begin to call *TopLevelWndFn* to process messages. This is the key to understanding *WinMain*'s message loop, which follows the call to *Init*.

The Message Loop

The message loop starts with a call to *GetMessage*. *GetMessage* retrieves a message from the application's message queue and fills the *msg* data structure with the message identifier and parameters. If the queue is empty, the call to *GetMessage* gives Windows' task manager a chance to transfer control to another task. The message loop thus serves a dual purpose: The application retrieves messages from its message queue, and Windows' task manager regularly regains control and allows other tasks to execute. The message loop continues to execute until *GetMessage* retrieves a WM_QUIT message from the application's message queue. For this message, *GetMessage* returns 0, which causes the loop and the application to terminate.

The message-loop functions *TranslateMessage* and *DispatchMessage* process each message that *GetMessage* retrieves from the message queue. The action of *TranslateMessage* is to translate the keyboard messages WM_KEYDOWN and WM_KEYUP into WM_CHAR messages. *TranslateMessage* also translates WM_SYSKEYDOWN and WM_SYSKEYUP messages (which are sent if the Alt key is down when another key is pressed) into WM_SYSCHAR messages. Although *TranslateMessage* is not strictly required for message processing, it is an essential component of the message loop in any program that expects keyboard input.

DispatchMessage performs the vital function of transferring control to a window function. *DispatchMessage* uses the window handle that is part of each message to determine the address of the corresponding window function. In the MODSTAT sample program, the window function is *TopLevelWndFn.* Each time the message loop in *WinMain* executes *DispatchMessage,* Windows calls *TopLevelWndFn* with the message identifier and its corresponding parameters.

GetMessage is one of the three Windows API functions that yield control to the task manager. The other functions—*PeekMessage* and *WaitMessage*—are most useful in a program that performs time-consuming or repetitive actions such as a prolonged computation or an I/O operation. *PeekMessage* is used in such a program instead of *GetMessage* because *PeekMessage* returns control to the program even if the message queue is empty. You can therefore use *PeekMessage* to design a message loop that frequently transfers control to Windows' task manager so that other applications can execute in the midst of a prolonged computation. The only constraint is that you split the prolonged computation into a series of time-limited steps, as the example in Figure 1-8 suggests.

```
bDone = FALSE;


do
{
  /* yield control and check the message queue */
  if( PeekMessage( &msg, ... ) )
  {
  /* process a message */
    if( WM_QUIT == msg.message )
      bDone = TRUE;

    else
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }
  }

  /* no message to process, so perform next step
       of computation */
  else
    ComputeNextPrimeNumber( ... );
}
while( !bDone );
```

Figure 1-8.
The main message loop in a Windows program that yields control in the midst of a repetitive computation. Compare this with the message loop in Figure 1-4.

The Window Function

In the sample application, *TopLevelWndFn* is the top-level window function, the function that processes all the messages that Windows sends to the application. *TopLevelWndFn* uses a *switch* statement to select from among four possible actions. The most complicated action occurs in response to a WM_PAINT message: *TopLevelWndFn* calls the private function *MsgPaint*, which in turn calls *ShowModuleInfo* to update the top-level window's client area with information about the application's module, task, and instance handles.

The only other messages that *TopLevelWndFn* processes are WM_COMMAND and WM_DESTROY. In each case, *TopLevelWndFn* takes an action that causes another message to be sent to the application. Although at first glance this might appear redundant, it is actually a common programming strategy in Windows applications. Often the most efficient way to cause a particular action to be carried out is to send a message to a window function that initiates the action.

For example, *TopLevelWndFn* processes WM_COMMAND by calling the private function *MsgCommand*, which checks whether the message's *wParam* value is IDM_ENUM. If it is, *MsgCommand* calls *InvalidateRect* to generate a WM_PAINT message in the application's message queue. When *TopLevelWndFn* subsequently processes the WM_PAINT message, the top-level window's client area is updated. In this way, the same function (*MsgPaint*) can be called in response to two different events: The window manager can send WM_PAINT whenever it determines that the client area needs to be repainted, or the user can generate WM_COMMAND with a *wParam* value of IDM_ENUM by choosing Enumerate from the application's main menu.

Similarly, *TopLevelWndFn* processes WM_DESTROY by calling the API function *PostQuitMessage*, which in turn places a WM_QUIT message in the application's message queue. When *GetMessage* subsequently retrieves this message, it returns 0 and causes the application's message processing to terminate.

Of course, *TopLevelWndFn* receives many other messages that it does not process explicitly. For these messages, the default action is to pass them to Windows' default message-processing function, *DefWindowProc*.

What MODSTAT Does

All the useful functionality of the sample application is encapsulated in two functions, *ShowModuleInfo* and *WndEnumFn*. *ShowModuleInfo* calls several Windows API functions to obtain information about the application module's status. It uses the API functions *wsprintf* to format its output and *TextOut* to write to the top-level window's client area.

ShowModuleInfo obtains information about other instances of the application by calling the API function *EnumWindows*. *EnumWindows* enumerates windows by calling *WndEnumFn* once for each overlapped and pop-up window in the window manager's list. When *WndEnumFn* is called for the top-level window of an instance of MODSTAT, the corresponding task and instance handles are displayed. The result is a display of the module's name, handle, and reference count, and a list of the task and instance handles for each instance of the application, as shown in Figure 1-9.

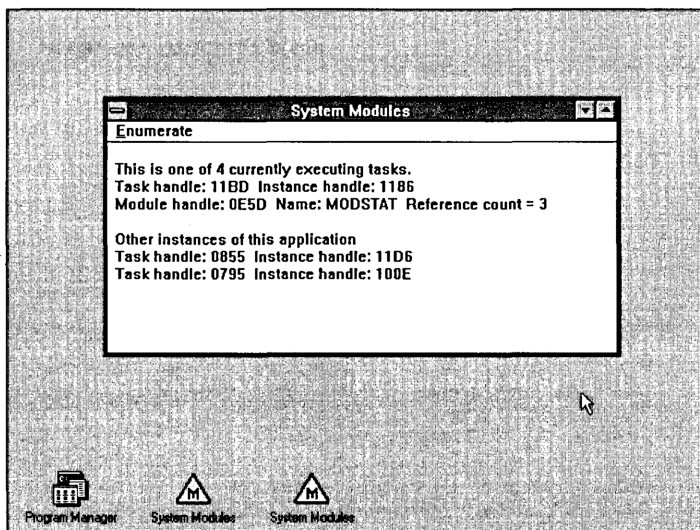


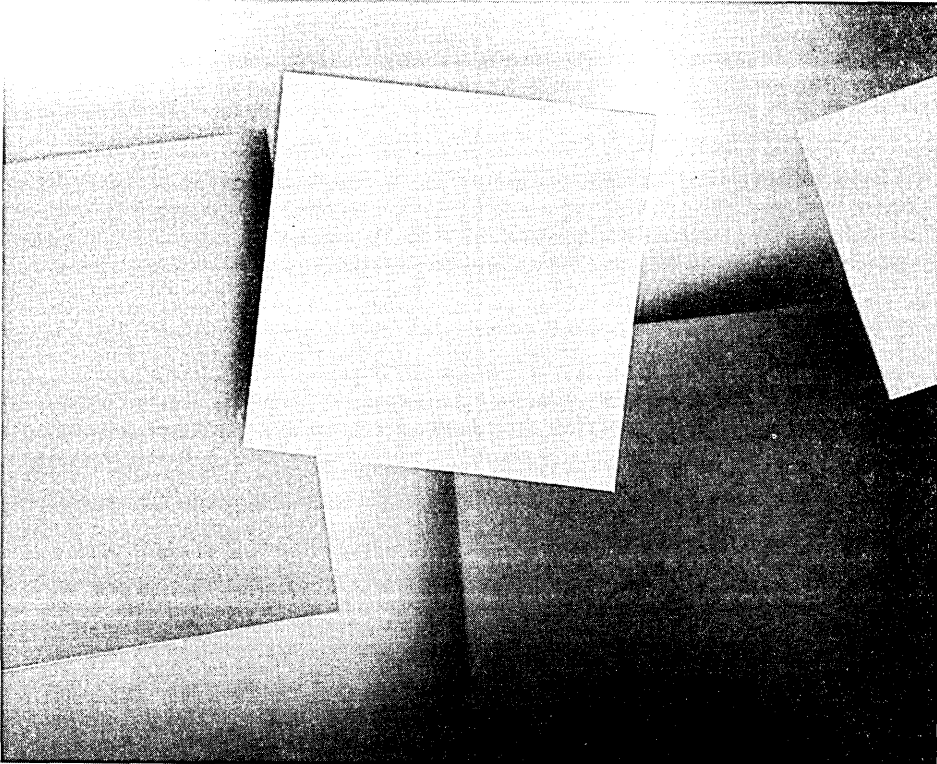
Figure 1-9.
Three instances of the sample application MODSTAT. Two of the instances are iconic; the third displays the task and instance handles of all three instances of the application.

On to Debugging

The structure of a Windows application reflects the design of the Windows environment. Applications that interact harmoniously with the Windows environment are the easiest to debug. The time you spend in careful structural design of your source code will be amply repaid in terms of the time you save in debugging.

2

Debugging



There is no secret way to debug a Windows program. You need the same combination of prevention, perseverance, and intuition to debug in Windows as you need in any other programming environment. In the Windows environment, you have a variety of debugging techniques and tools at your disposal. Some of these are unique to Windows, whereas others are simply traditional programming techniques with a Windows twist. The key to efficient debugging, however, is not a technique at all—it is a well-designed program.

Designing for Debugging

The principle that good software design leads to easier debugging holds true in Windows. In general, an easy-to-debug Windows program has a structure that reflects the way the program interacts with the Windows environment. For example, you may want to isolate different window functions in different units of source code so that the functionality of each window in a program is encapsulated in its own unit of source code. There are also a few specific details of source-code design that will help make your Windows programs easier to debug. These include simple source-code statements, modular design, and accurate use of functions and variables.

Use Simple Source-Code Statements

When you use one of the debuggers supplied with the Microsoft Windows SDK, the best you can do when you find a bug is to associate it with a particular line of source code. You can more easily pinpoint a bug if you use simple source-code statements and limit yourself to only one statement in each line of source code.

Modularize Your Source Code

Modular source code simplifies debugging. This is especially true in Windows, where initialization code, message dispatching loops, and window functions fall naturally into separate blocks of source code. Often the quickest way to zero in on a bug is to recognize which block of source code is malfunctioning.

When you use one of Microsoft's Windows debuggers—CodeView, SYMDEB, or WDEB386—you can benefit from keeping a program's source code in separate files. If you use the medium memory model to compile your applications, the Microsoft C compiler will assign each source-code file's name to the corresponding executable-code segment. When a debugger subsequently traps a bug, it uses the name of the segment and an offset into the segment to locate the bug in the executable code. You can then work backward from the segment name and offset to a particular line of source code in the corresponding file.

Use Function Prototypes

Some of the toughest bugs to fix are those that occur when the C compiler makes invalid assumptions about the data types associated with the parameters in a function call. You can avoid this kind of error by incorporating function prototypes throughout your source-code design. For functions used only within a single source-code module, place your function prototypes near the beginning of the module, before you define any of the prototyped functions. For functions that are used globally, place function prototypes in an include file and include the file after WINDOWS.H in any source-code module that makes global calls.

Avoid Static and Global Variables

Another tough bug to uncover in a Windows application occurs when a window function stores data in a static or global variable. Typically, the value is something that is used only within a particular window, such as a GDI handle or a pointer to a window function.

The problem with using a static or global variable is that only one window at a time can store data in the variable. If two windows in your application store data in the same static or global variable, one window must lose its data. You can avoid this bug by storing window-specific data in a window's extra bytes or property list rather than in a static variable. (There is more information about window extra bytes and property lists in Chapter 5.)

You should use a static or a global variable only if you are certain that only one window at a time will ever access the variable. Nevertheless, a safer technique is to reserve static and global variables for values that are truly global and not private to a particular window.

Using a Debugging Terminal

One way to save time when debugging a Windows application is to use a debugging terminal to display debugging data while the application executes. A debugging terminal consists of a dumb terminal (keyboard and screen) connected to your computer's serial communications port through a null-modem cable. You can also use a second computer and a terminal-emulation program instead of a dumb terminal. If your computer system contains two different video subsystems, you can emulate a debugging terminal by using one video subsystem for Windows and the other for displaying debugging data. This method requires you to use a special device driver that redirects serial output to the second video subsystem. Such device drivers are in the public domain and can be downloaded from bulletin-board systems such as CompuServe.

The reason to use a debugging terminal is so that you need not rely on the Windows screen to view debugging data. If you try to debug entirely within the confines of the Windows display, you will find that your application can't get out of its own way when it tries to display debugging information. The bug that crashes your application might prevent the application from displaying the debugging message that would help you identify the bug.

You can make good use of a debugging terminal with the tools provided in the Windows SDK. The Windows API supports a function called *OutputDebugString*, which sends a string to the debugging terminal. Furthermore, when the debugging version of Windows traps an error that causes a fatal exit or when an application detects an error and calls the *FatalExit* API function, Windows displays the error number and a stack trace on the debugging terminal. Also, all three of Microsoft's Windows debuggers support the use of a secondary monitor, a debugging terminal, or both.

Windows Debuggers

Three debuggers are supplied with the Windows SDK: CodeView for Windows, SYMDEB, and WDEB386. Although all three use a similar set of debugging commands, each gives you a different view of the Windows environment. The debugger you use for a particular application depends on your debugging needs.

If you want to debug in protected mode, use CodeView or WDEB386. To debug in real mode, use SYMDEB. Use WDEB386 if you need access to descriptor tables, I/O ports, and other low-level components of the Windows environment. All three debuggers let you debug a program at the level of assembly language, but if you want to use a debugger that can display C source-code statements, you must use CodeView or SYMDEB.

CodeView

Of the three SDK debuggers, CodeView is probably the best suited to Windows application debugging. CodeView lets you view an application's C source code, set breakpoints, and trace through the source code as the program executes. Also, CodeView is the only one of the three debuggers that lets you monitor the flow of window messages and set breakpoints on specific messages. Sometimes the quickest way to home in on a mysterious bug is to use CodeView's *wwm* (Windows watch message) command on all of an application's window functions to determine which message causes the bug to appear.

Before you use CodeView, you must compile and link your application with special command-line switches, as shown in Figure 2-1. The */Od* switch disables compiler optimization that could interfere with debugging. The */Zi* and */CO* switches cause line numbers and symbols (function and variable names) to be incorporated into the application's executable file for CodeView's use. This extra debugging information does not affect the flow of control of the application. CodeView uses the symbols and line numbers to associate names with their corresponding memory locations and to display source-code statements as the program executes.

Switch	***Program***	***Comments***
/Od	Microsoft C Compiler	Disables all compile-time optimizations
/Zi	Microsoft C Compiler	Includes line-number, global-name, and local-name information in compiled code
/CO	Microsoft Linker	Includes line-number, global-name, and local-name information in executable file

Figure 2-1.
Compiler and linker switches for preparing a program for debugging with CodeView.

SYMDEB

SYMDEB (a SYMbolic DEBugger) provides a superset of the functionality supported in the MS-DOS debugger, DEBUG. Like DEBUG, SYMDEB lets you inspect data in memory. It supports breakpoints at the assembly-language level and lets you view both assembly-language instructions and the corresponding C-language source-code lines.

You need a debugging terminal to use SYMDEB. If your computer system contains two video subsystems, one of which is a monochrome display adapter, SYMDEB can use the monochrome screen as a debugging terminal. You start SYMDEB by running it on your primary display. To redirect output to a monochrome display, specify the */m* switch on the SYMDEB command line. If you're using a debugging terminal, redirect SYMDEB's output to the debugging terminal with SYMDEB's *=COM1* or *=COM2* command. You begin debugging by setting breakpoints and source-code display options. Then you let SYMDEB execute your Windows application.

To prepare an application for debugging with SYMDEB, use the compiler and linker switches listed in Figure 2-2, and use the MAPSYM utility to create a .SYM (symbol) file from the .MAP file for the application.

Switch	Program	Comments
/Od	Microsoft C Compiler	Disables all compile-time optimizations
/Zd	Microsoft C Compiler	Includes line-number and global-name information in compiled code
/MAP	Microsoft Linker	Creates .MAP file
/LI	Microsoft Linker	Includes line-number information in .MAP file

Figure 2-2.
Compiler and linker switches for preparing a program for debugging with SYMDEB or WDEB386.

SYMDEB does not support a full-screen interface as CodeView does. Also, SYMDEB runs only in real mode, not in protected mode. Despite these drawbacks, there are certain debugging situations in which you might need SYMDEB. One is when you suspect that a bug in your application occurs as a result of Windows' memory manager moving a block of global memory. Because the pointers used to access global-memory blocks can be invalidated only in real mode, SYMDEB is the best tool for finding such a bug. Also, if you suspect that a bug is related to an application's use of expanded (EMS) memory, you can use SYMDEB to monitor expanded-memory activity as the application runs.

WDEB386

If you want to debug in protected mode, an alternative to using CodeView is to run WDEB386 with a debugging terminal. Despite its name, WDEB386 runs on both 80286 and 80386 CPUs. WDEB386 lacks the convenient features of CodeView, such as a full-screen interface and the ability to debug source code as well as assembly-language code. On the other hand, WDEB386 gives you access to elements of the programming environment that are not available through CodeView, including the contents of the global and local descriptor tables. This feature makes WDEB386 particularly useful to device-driver developers and to assembly-language programmers.

You can also use WDEB386 to debug Windows applications that are written in C. Prepare an application for debugging with WDEB386 the same way you do for SYMDEB. Use the compiler and linker switches listed in Figure 2-2, and create a .SYM file for the application by using MAPSYM. If your source code is contained in more than one C source file, you might also want to use the */AM* switch with the Microsoft C compiler to build the application with the medium memory

model. This memory model may be easier to use when you work backward from an assembly-language instruction in WDEB386 to a line of source code in a C source file. You might also use the */Fc* switch, which produces a .COD file that lists both source code and the corresponding compiler-generated assembly language.

When you load WDEB386, the command-line options you use depend on the debugging environment. Use the */C:* switch to specify which serial communications port WDEB386 will use and the */S:* switch to specify symbol files. With an 80286-based computer, use KRNL286.SYM and the */s* (standard mode) switch:

```
WDEB386 /C:1 /S:krnl286.sym /S:myapp.sym win.com /s myapp
```

In an 80386 environment, specify KRNL386.SYM and either the */s* (standard mode) or the */3* (enhanced mode) switch:

```
WDEB386 /C:1 /S:krnl386.sym /S:myapp.sym win.com /3 myapp
```

Figure 2-3 illustrates how WDEB386 might be used to find a bug that has produced an "Unrecoverable Application Error" message in an application called MYAPP. When WDEB386 starts, it lets the Windows kernel run long enough to establish the protected-mode environment. It then halts at a breakpoint (INT 3) that is built into the Windows kernel. This breakpoint exists only to return control to the debugger so that you can set breakpoints in your application and adjust WDEB386 display options.

In Figure 2-3, three commands are entered at the # prompt after this breakpoint is encountered. The *y 386env* command toggles the *386env* flag, which causes WDEB386 to display only the 16-bit CPU registers. The *z* command replaces the INT 3 breakpoint instruction with a NOP so that it won't be encountered again. Then the *g* command transfers control to the application.

When the application encounters the bug, the familiar "Unrecoverable Application Error" message box appears on the Windows screen. Selecting the *Cancel* button in the message box returns control to WDEB386, which displays the assembly-language instruction that caused the error. In Figure 2-3, the instruction is *MOV AL, BYTE PTR ES:[BX+SI]*. Because the instruction accesses memory, it's a good bet that the cause of the error is an attempt to access a protected memory location. The *dl es* command, which displays the ES entry in the Local Descriptor Table, verifies that this is the case. The selector in ES (105DH) is a valid data-segment selector because the *dl es* command successfully displayed a descriptor-table entry. However, the largest valid offset in the segment (that is, the segment limit) is 000FH. Because the offset [BX+SI] is 0010H, the error is that the program has attempted to read a byte that lies outside the data segment.

```
Microsoft (R) Windows 3.0 Kernel Debugger  Version 2.75  09.Mar.90
Copyright (C) Microsoft Corp 1990. All rights reserved.
[80286]
Map linked (KERNEL)
Map linked (USER)
Map linked (GDI)
Map linked (MYAPP)
DOSX!DATA(0000)=3ACF
DOSX!CODE(0296)=3D65
DOSX!CODE(04CF)=0050
KERNEL!IGROUP=02D5
KERNEL!_NRESTEXT=02DD
KERNEL!_MISCTEXT=02E5
SYSTEM!CODE(0001)=036D
:
MSDOS!CODE(0001)=0F45
MSDOS!CODE(0002)=0F4D
MSDOS!CODE(0003)=0F55
MSDOS!DATA(0006)=0F75
AX=000042F0 BX=0000138F CX=00000000 DX=00000006 SI=00000E08 DI=00000000
IP=0000827E SP=00001A76 BP=00001A98 CR2=00000000 CR3=00000 IOPL=3  F=-- --
CS=02D5 SS=0F75 DS=0081 ES=0DF5 FS=0000 GS=0000 -- NV UP EI NG NZ NA PO NC
02D5:0000827E  INT     3
#y 386env
#z
02D5:827E  INT     3  replaced with:  NOP
#g
MYAPP!_TEXT=06AD
MYAPP!DGROUP=069D
MYAPP!MYAPP_TEXT=0E4D
AX=0000  BX=0000  CX=105D  DX=105D  SP=1514  BP=1524  SI=0010  DI=104E
IP=036C  CS=0E4D  DS=069D  ES=105D  SS=069D  -- NV UP EI NG NZ AC PE CY
0E4D:036C  MOV     AL,BYTE PTR ES:[BX+SI]                      ES:0010=INV:0003
#dl es
105D  Data    Bas=04AB70 Lim=000F DPL=1 P  RW    A
#ln
0E4D:01D0 MYAPP!MYAPP_TEXT:WndFn + 19C
```

Figure 2-3.
A sample debugging session in WDEB386.

You can find the bug in the C source code by using the location of the assembly-language instruction that caused the error. (In Figure 2-3, the segment:offset location of the instruction is 0E4D:036C.) The *ln* command shows that the code segment that contains the instruction is named MYAPP_TEXT. You can then refer to the list of line numbers in MYAPP.MAP that correspond to MYAPP_TEXT as shown in Figure 2-4. From this list you can see that the value of the offset,

036CH, lies somewhere in the executable code that corresponds to line 323 of the source code. Finally, a look at line 323 in MYAPP.C in Figure 2-5 reveals the bug: The array subscript *n* is larger than the size of the global-memory block to which *pData* points.

```
Line numbers for myapp.obj(myapp.c) segment MYAPP_TEXT

    62 0001:0000    67 0001:000D    68 0001:0020    69 0001:0024
    72 0001:0026    74 0001:003B    75 0001:0045    78 0001:004F
    79 0001:0066    89 0001:0070    90 0001:007A    91 0001:0080
    93 0001:0089    95 0001:0095    96 0001:00A1   107 0001:00A9
   112 0001:00B6   113 0001:00BE   114 0001:00C4   115 0001:00CE
   116 0001:00DD   117 0001:00EC   118 0001:00F4   119 0001:00F9
   120 0001:00FE   121 0001:0103   123 0001:0108   124 0001:0112
   135 0001:011A   136 0001:0124   137 0001:0131   154 0001:0139
   161 0001:0146   176 0001:014C   179 0001:0174   180 0001:017D
   181 0001:0188   183 0001:0193   184 0001:0199   187 0001:01A2
   188 0001:01AD   190 0001:01B5   192 0001:01C5   193 0001:01C8
   203 0001:01D0   204 0001:01DA   223 0001:01ED   207 0001:0203
   208 0001:0216   211 0001:0218   212 0001:0228   215 0001:022A
   216 0001:0232   219 0001:0234   227 0001:023B   228 0001:023E
   238 0001:0248   239 0001:0252   242 0001:0262   243 0001:026A
   246 0001:026C   247 0001:0274   250 0001:0276   256 0001:027F
   266 0001:0287   267 0001:0291   277 0001:0299   281 0001:02A6
   282 0001:02BB   283 0001:02CD   284 0001:02D8   294 0001:02E0
   300 0001:02ED   302 0001:02FD   303 0001:0306   305 0001:030F
   307 0001:0321   308 0001:032E   313 0001:0336   319 0001:0344
   320 0001:0352   322 0001:035E   323 0001:0366   325 0001:037C
   326 0001:0384   328 0001:038C   329 0001:038F
```

Figure 2-4.
Line numbers and offsets in a .MAP file produced by MAPSYM. If you know a segment name and offset, you can use this table to find the corresponding source-code line number.

```
Line

312  static int Bomb( HWND hWnd )
313  {
314    LPSTR        pData;
315    GLOBALHANDLE hData;
316    int          i,n;
```

Figure 2-5. *(continued)*
An excerpt from source code that contains a memory-protection bug.

Figure 2-5. *continued*

```
317
318
319     hData = GlobalAlloc( GHND, 16L );
320     pData = GlobalLock( hData );
321
322     for( n=0; n<20; n++ )
323       i += pData[n];
324
325     GlobalUnlock( hData );
326     GlobalFree( hData );
327
328     return i;
329   }
```

Useful Debugging Techniques

Along with the assortment of debugging methods provided by Windows debuggers, there are several programming techniques that you can use to prevent and detect errors in your Windows applications. You should choose the debugging techniques best suited to the kind of bug you are hunting and to your personal debugging style.

Using the Debugging Version of Windows

The foremost rule of thumb in debugging a Windows application is this: Always use the debugging version of Windows to test your application. The debugging version notifies you of errors that might pass undetected in the retail Windows version. For example, if an application calls *LocalFree* to free a block of memory that has been locked (by calling *LocalLock*) but not unlocked (by calling *LocalUnlock*), the debugging version of Windows interrupts the application with an error message and a prompt to abort (terminate the application), ignore (continue execution despite the error), or break (give control to the debugger). If you used the retail version of Windows, this error would pass unnoticed.

Scaffolding

Scaffolding is a time-honored debugging technique that consists of embedding extra debugging source code in a program. The purpose of scaffolding is to track the state of the program at key locations in the source code. Here are some of the many ways to use scaffolding:

- To view the intermediate results of computations.
- To monitor variables whose values control what is drawn on the screen.

- To record a program's flow of control.
- To test sections of source code that might otherwise execute only occasionally.

The most effective way to incorporate scaffolding into an application is to compile it conditionally. For example, Figure 2-6 contains code that calls *OutputDebugString* to display the values of the *hWnd*, *hData*, and *pData* variables. The *#ifdef* and *#endif* preprocessor directives bracket the scaffolding code so that it is compiled into the application only if the symbol *DEBUG* is defined at compile time. (You can use either a *#define* directive in the source code or the */D* switch on the Microsoft C compiler's command line to define the symbol *DEBUG*.) This technique lets you include or omit debugging code, without modifying the actual source code, by recompiling with or without the appropriate preprocessor symbol definitions. You can use this technique to create levels of debugging code that provide different amounts of debugging output by nesting conditionally compiled code using two or more preprocessor symbols.

```
void TestFn( HWND hWnd, GLOBALHANDLE hData )
{
  LPSTR    pData;

#ifdef DEBUG
  char     DebugString[64];

  wsprintf( DebugString, "TestFn: hWnd=%04X, hData=%04X, pData=%08lX\r\n",
            hWnd, hData, pData );
  OutputDebugString( DebugString );
#endif

  pData = GlobalLock( hData );
  MessageBox( hWnd, pData, "Test", MB_OK );
  GlobalUnlock( hData );
}
```

Figure 2-6.
Conditionally compiled scaffolding (debugging code) in a Windows function.

Scaffolding techniques complement the use of a debugger to single-step through source code and to set breakpoints or watchpoints. The Windows API function *OutputDebugString* is designed to be used with a debugger as well as with a debugging terminal. If you're using CodeView, for example, *OutputDebugString* directs its output to CodeView's command window.

Tracing Messages

Because flow of control in a Windows program is governed by messages, one of the most productive ways to debug a Windows application is to trace the flow of messages. If a bug appears in response to a particular message, a good place to look for the error is in the source code that processes the message.

There are at least three ways to trace messages. The easiest is to use the Spy utility in the Windows SDK, which lets you use the mouse to select a window and then shows you the window's messages as they are processed. However, Spy can spy only on visible windows. You can't use Spy to trace a window's messages if the window isn't visible somewhere on the screen.

A more selective approach is to use CodeView, which lets you debug your application as well as trace messages. CodeView's *wwm* (Windows watch message) command lets you trace specified messages in multiple window functions as well as in multiple instances of the same window. Figure 2-7 shows how CodeView displays the first few messages sent to the top-level window of an application.

```
HWND:1340 wParam:0000 lParam:0C8D07C4 msg:0024 WM_GETMINMAXINFO
HWND:1340 wParam:0000 lParam:12351524 msg:0081 WM_NCCREATE
HWND:1340 wParam:0000 lParam:1235150A msg:0083 WM_NCCALCSIZE
HWND:1340 wParam:0000 lParam:12351524 msg:0001 WM_CREATE
HWND:1340 wParam:0001 lParam:00000000 msg:0018 WM_SHOWWINDOW
```

Figure 2-7.
Sample message trace produced by using CodeView's wwm *command.*

CodeView's message-tracing ability is particularly useful when you combine it with the *wbm* (Windows break message) command, which lets you set a breakpoint on a particular message. One way to locate a bug in a Windows program is to use *wwm* to trace messages until the bug appears. If the bug seems to be associated with a particular message, you can use *wbm* to set a breakpoint on the message and then single-step through the code that processes the message.

A third message-tracing technique is to embed message-decoding scaffolding in an application's window functions. One way to do this is to write a function that uses a *switch* statement to decode a message's *wMsg*, *wParam*, and *lParam* parameters. The function should use *OutputDebugString* to display the message contents. This technique lets you trace specific messages and parameters more informatively than you can with CodeView or Spy.

Intercepting API Functions

A clean way to encapsulate debugging code is to construct a set of functions that intercept calls to Windows API functions. Each intercept function can surround a Windows API call with error checking and other scaffolding.

Figure 2-8 shows how you might design a set of intercept functions for the Windows API functions that access window extra bytes. The four functions *xGWW*, *xSWW*, *xGWL*, and *xSWL* intercept calls to the *GetWindowWord*, *SetWindowWord*, *GetWindowLong*, and *SetWindowLong* Windows API functions. The functions trap any attempts to access or set window extra bytes that are not allocated for the specified window.

```
static char szMsg[80];
static char szOOB[] = " out of bounds\r\n";

/*************************************************************************
 *                                                                       *
 * xGWW--Intercepts GetWindowWord                                        *
 *                                                                       *
 *************************************************************************/

WORD PASCAL FAR xGWW( HWND hWnd, int nOffset )
{
    WORD    wRVal;

    if( IsOK( hWnd, nOffset, sizeof(WORD) ) )
      wRVal = GetWindowWord( hWnd, nOffset );

    else
    {
      MessageBeep( 0 );
      wsprintf( szMsg, "GetWindowWord( %04X, %d )", hWnd, nOffset );
      lstrcat( szMsg, szOOB );
      OutputDebugString( szMsg );

      wRVal = 0;
    }
    return wRVal;
}
```

Figure 2-8. *(continued)*

Intercept functions for debugging calls to GetWindowWord, SetWindowWord, GetWindowLong, *and* SetWindowLong.

Figure 2-8. *continued*

```
/************************************************************************
*                                                                       *
* xSWW--Intercepts SetWindowWord                                        *
*                                                                       *
************************************************************************/

WORD PASCAL FAR xSWW( HWND hWnd, int nOffset, WORD wNew )
{
    WORD    wRVal;

    if( IsOK( hWnd, nOffset, sizeof(WORD) ) )
      wRVal = SetWindowWord( hWnd, nOffset, wNew );

    else
    {
      MessageBeep( 0 );
      wsprintf( szMsg, "SetWindowWord( %04X, %d, %u )",
        hWnd, nOffset, wNew );
      lstrcat( szMsg, szOOB );
      OutputDebugString( szMsg );

      wRVal = 0;
    }
    return wRVal;
}

/************************************************************************
*                                                                       *
* xGWL--Intercepts GetWindowLong                                        *
*                                                                       *
************************************************************************/

LONG PASCAL FAR xGWL( HWND hWnd, int nOffset )
{
    LONG    lRVal;

    if( IsOK( hWnd, nOffset, sizeof(LONG) ) )
      lRVal = GetWindowLong( hWnd, nOffset );

    else
    {
      MessageBeep( 0 );
      wsprintf( szMsg, "GetWindowLong( %04X, %d )", hWnd, nOffset );
      lstrcat( szMsg, szOOB );
      OutputDebugString( szMsg );

      lRVal = 0L;
    }
```

(continued)

Figure 2-8. *continued*

```
    return lRVal;
}

/****************************************************************************
*                                                                           *
* xSWL--Intercepts SetWindowLong                                            *
*                                                                           *
****************************************************************************/

LONG PASCAL FAR xSWL( HWND hWnd, int nOffset, LONG lNew )
{
    LONG    lRVal;

    if( IsOK( hWnd, nOffset, sizeof(LONG) ) )
      lRVal = SetWindowLong( hWnd, nOffset, lNew );

    else
    {
      MessageBeep( 0 );
      wsprintf( szMsg, "SetWindowLong( %04X, %d, %lu )",
        hWnd, nOffset, lNew );
      lstrcat( szMsg, szOOB );
      OutputDebugString( szMsg );

      lRVal = 0L;
    }
    return lRVal;
}

/****************************************************************************
*                                                                           *
* IsOK--Tests whether specified space is allocated                          *
*                                                                           *
****************************************************************************/

static BOOL IsOK( HWND hWnd, int nOffset, int nSize )
{
    int     nEB;
    BOOL    bRVal = FALSE;

    if( nOffset >= 0 )
    {
      nEB = GetClassWord( hWnd, GCW_CBWNDEXTRA );
      bRVal = (nOffset + nSize) <= nEB;
    }
    return bRVal;
}
```

Although you can incorporate intercept functions directly into a program's source code, a more flexible strategy is to build intercept functions into a dynamic link library. (Chapter 3 contains more information on dynamic link libraries.) In the library's module-definition (.DEF) file, assign each intercept function an export name that matches the name of the corresponding API function, as shown in Figure 2-9.

```
EXPORTS   WEP                          @1 RESIDENTNAME
          GetWindowWord = xGWW @133
          SetWindowWord = xSWW @134
          GetWindowLong = xGWL @135
          SetWindowLong = xSWL @136
```

Figure 2-9.
Exporting intercept functions in the module-definition file of a dynamic link library. The exported names are identical to the corresponding functions in the Windows API.

After you compile the dynamic link library, use IMPLIB to create an import library for the intercept functions. You can then debug an application by linking the application with the DLL's import library before the standard Windows API library (LIBW.LIB):

```
LINK /NOD /NOE myapp, , , intercpt libw, myapp
```

Placing the reference to the import library (in this example, INTERCPT.LIB) ahead of the reference to LIBW.LIB causes the application's API calls to be linked to the intercept functions in INTERCPT.DLL instead of the default Windows API functions. However, the intercept functions themselves will be linked to the default API functions. When you execute the application, the intercept functions will be called. When you finish debugging, you can simply recompile the application without reference to INTERCPT.LIB, so all the application's API calls will be linked directly to the default Windows functions.

Common Bugs

The most commonly encountered bugs in Windows applications are related to the nature of the Windows environment. Windows applications run in a visually rich, interactive system where programming errors quickly become visible on the screen. Also, Windows' memory-management strategy, in which blocks of memory are managed dynamically outside the control of an application, can unmask a number of subtle programming errors.

Visible Bugs

It is usually easy to find bugs that affect the appearance of a window. If you create a window with the wrong style or initialize the window's background color with the wrong value, the visual appearance of the window will clue you in to the problem in your source code. If your list-box controls don't list or your scroll-bar controls don't scroll, a good place to look for a bug is in the source code responsible for manipulating and responding to those controls.

A trickier bug to discover is one that causes a window never to appear on the screen at all. Here are several reasons why this may happen:

- The window function is not exported.
- An invalid window class name or instance handle is specified in the WNDCLASS data structure when *RegisterClass* is called.
- The window class name or parent-window handle specified in the *Create-Window* function is invalid.
- The window function does not pass unprocessed messages to another window function such as *DefWindowProc.*
- The window does not have the WS_VISIBLE style, or calls to *ShowWindow* or *UpdateWindow* do not execute successfully.
- The window is overlapped by another window.
- The window has no border, and its class background color is the same as the desktop (COLOR_BACKGROUND) or the parent's client area (COLOR_WINDOW).

You can avoid some of these problems by checking the return values from functions such as *RegisterClass* and *CreateWindow.* In other cases, the only way to find the bug is to use a debugger or appropriate scaffolding to examine the WNDCLASS data structure, window handle, and window style bits.

Confusing Static and Automatic Data

Another type of bug is related to the notion of static and automatic data elements in the C programming language. Storage space for static data elements is allocated in a program's default data segment. Thus the space reserved for static data remains allocated as long as your program executes. In contrast, memory for automatic data elements is allocated on the stack at the time control is transferred to the function in which the automatic variables are declared. When the function exits, the stack memory is reclaimed, and the values of the automatic data elements are lost.

This kind of bug can appear in a window function if the function stores a value in an automatic variable in response to a message and then attempts to use that value in response to a subsequent message. In Figure 2-10, for example, *hBrush* is declared as an automatic variable. Because memory for *hBrush* remains allocated only until *WndFn* returns, the value stored in *hBrush* when the message WM_CREATE is processed is no longer available when WM_CTLCOLOR is processed. The bug can be fixed by declaring *hBrush* as a static variable or by storing the brush handle in the window's extra bytes or property list.

```
LONG PASCAL FAR
WndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
  HBRUSH   hBrush;     /* hBrush is an automatic variable */

  switch( wMsg )
  {
    case WM_CREATE:
      hBrush = GetStockObject( GRAY_BRUSH );
      break;

    case WM_CTLCOLOR:
      return MAKELONG( hBrush, 0 ); /* ERROR: hBrush isn't valid */
      break;
    :
```

Figure 2-10.
*A bug caused by using an automatic variable (*hBrush*) to store a value across multiple messages.*

Fatal-Exit Errors

The debugging version of Windows is designed to catch a variety of common bugs. When a bug occurs, the debugging version displays an error number that indicates the nature of the error. Appendix C of the reference manual in the Windows SDK documentation contains a complete list of fatal-exit error codes.

Following the error number, the debugging kernel produces a stack trace that lists the return addresses for the function calls that occurred prior to the error. This information appears on the debugging terminal unless you're using CodeView, which displays it in the command window.

Figure 2-11 on the following page shows a typical fatal-exit stack trace caused by an application's attempt to use an invalid value for a global-memory handle. The topmost return address is the most recent. This list of addresses may seem cryptic, but if you use one of the Windows debuggers, you can work backward through the stack trace to the location in your source code that caused the error.

```
gdref: invalid handle 0000:3302

FatalExit code = 0x0280
Stack trace:
02D5:7BBA
02D5:1F56
12A5:0352
12A5:0278
12A5:020E
0BB5:1005
12A5:0032
12B5:0077

Abort, Break or Ignore?
```

Figure 2-11.
A stack trace displayed in CodeView's command window by the Windows debugging kernel when a fatal error is detected. The error number (0x0280) indicates that the error resulted from an attempt to use an invalid global-memory handle. The return addresses on the stack are in selector:offset form.

For example, you can use the *v* (view) command in CodeView to work backward from a return address in the stack trace to the source-code line where the error occurred. In the stack trace shown in Figure 2-11, the first two addresses in the list correspond to function calls made outside the application. (Entering *v 0x02d5:0x7bba* or *v 0x02d5:0x1f56* in the command window produces the message "No source lines at this address.") The next return address, however, does point within the application. Entering *v 0x12a5:0x0352* moves the cursor to an assembly-language instruction that corresponds to line 320 of the application's C source code, as shown in Figure 2-12. This is where the fatal error occurred. The bug occurs because the automatic variable *hData* was never initialized, as it probably should have been in line 319.

```
318:
319:            GlobalAlloc( GHND, 16L );
12A5:0344 6A42          PUSH      42
12A5:0346 6A00          PUSH      00
12A5:0348 6A10          PUSH      10
12A5:034A 9AE206D502    CALL      02D5:06E2
320:            pData = GlobalLock( hData );
12A5:034F FF76FC        PUSH      Word Ptr [BP-04]
12A5:0352 9AA009D502    CALL      02D5:09A0
12A5:0357 8946F6        MOV       Word Ptr [BP-0A],AX
12A5:035A 8956F8        MOV       Word Ptr [BP-08],DX
```

Figure 2-12.
A view of the combined source/assembly listing in CodeView's source window, showing where the fatal error in Figure 2-11 occurred.

Wild Pointers

A wild pointer is a pointer to data in the wrong part of memory. When your program uses a wild pointer to read from or write to a memory location, almost anything can happen. If a wild pointer causes data in a program's default data segment to be corrupted, you might be able to locate the source of the corrupted data very quickly. On the other hand, if a wild pointer points to a block of data owned by Windows itself, the program that crashes as a result of the corrupted data may not be the same as the one that contains the wild pointer. In such cases, a wild-pointer hunt may seem more like a wild-goose chase.

The reason wild pointers can be hard to isolate is that their bad effects often become apparent in misleading ways. As the result of a wild pointer, an application might terminate suddenly with only Windows' "Unrecoverable Application Error" message box to indicate that something went wrong. A wild pointer can also crash a program in a confusing way, by causing a spurious fatal-exit error such as "invalid window handle" (error code 0x0007) or "gdref: invalid handle" (error code 0x0280). Such errors don't represent what they seem to. They occur because your application has somehow clobbered data that actually belongs to Windows' window manager or memory manager. If you see a fatal-exit error that seems unrelated to what a program is supposed to be doing, you can suspect a wild pointer.

Most wild pointers are the result of "dumb" mistakes, unrelated to errors in program design or execution logic. Wild pointers can creep into a Windows application when you forget to initialize a pointer variable, make an error in pointer arithmetic, use a pointer to a memory block that has moved to another location, or fail to properly associate an exported function with its default data segment.

Uninitialized pointer variables

If you dereference a pointer without first specifying its value, you are almost certainly accessing the wrong part of memory. This is an easy mistake to make when you program in C because the C language lets you use the same syntax to dereference both pointer variables and array names. Consider the following example:

```
void UninitializedPointer()
{
  LPSTR    Buf;

  Buf[0] = 0;    /* ERROR: Buf is uninitialized! */
}
```

In this function, the assignment statement contains a bug because the pointer *Buf* is uninitialized. The programmer's intention might have been to declare *Buf* not as an *LPSTR* but as a character array:

```
char    Buf[50];
```

This kind of wild pointer is usually easy to trap when you debug in protected mode. There is only a small chance that the value that happens to be stored in an uninitialized pointer variable is actually a valid selector:offset combination, so CPU memory protection usually traps the error. When the error occurs, you'll see the "Unrecoverable Application Error" message box. With a debugger, you can view the assembly-language instruction that used the wild-pointer value.

Errors in pointer arithmetic

Another kind of wild pointer results from an error in pointer arithmetic. For example, the following loop addresses elements in an array that are not allocated in the array declaration:

```
int     n, x[100];

for( n = 0; n < 200; n++ )
  x[n] = n;
```

Unfortunately, there is no way to check for pointer-arithmetic errors automatically during a program's execution. Your best bet is to debug in protected mode and hope that CPU memory-protection traps any errors in pointer arithmetic.

Clobbered data

When they don't cause memory-protection errors, pointer-arithmetic bugs generally lead to clobbered data. The way these bugs appear depends on the data you clobber. On the stack, you might overwrite a return address, which would cause a program to fail as soon as it tried to use the return address to return from a function call. You might also destroy the stack-frame pointer saved on the stack each time a function is called. This could lead to fatal-exit error 0x0303 ("invalid BP chain"). In the local heap, you might overwrite one of the linked-list pointers Windows memory manager uses to maintain the local heap. In this case, you might see fatal-exit errors 0x0100, 0x0103, or 0x0140, all of which occur when memory management of the local heap is disrupted.

Another way in which you might accidentally overwrite data is in calls to the API functions *SetWindowWord* and *SetWindowLong*, which are used to access window extra bytes. You are certain to have problems if you specify the wrong byte offset or if you use *SetWindowLong* when you should have used *SetWindowWord.* You will also corrupt data if you do not specify a large enough value for the *cbWndExtra* field of the WNDCLASS data structure when you register a window class.

This kind of bug can cause an application to fail in a misleading way because the data being overwritten is used by Windows' window manager. For example, you might see fatal-exit 0x0007 ("invalid window handle") or even 0x0140 ("Local heap is busy"), when the real problem is that you used a window's extra bytes incorrectly. You can also crash other applications by attempting to store data in window extra bytes beyond what you allocate in the WNDCLASS data structure. This happens because window extra bytes are part of a data structure in a heap maintained by Windows' window manager on behalf of all windows in all applications. If a program stores data beyond the allocated number of window extra bytes, it may corrupt the heap and disrupt all window management.

Dissociated data segments

In Windows, every instance of every application is associated with a different default data segment. Whenever control transfers to an application instance—that is, whenever an exported far function is called—Windows loads the CPU's DS register with the segment value (in real mode) or selector (in protected mode) that identifies the instance's default data segment. If this doesn't happen, all sorts of wild-pointer errors can occur because the value in DS may not point to the default data segment associated with the instance.

To understand why this is so, consider how Windows stores the appropriate default data-segment value in DS. In Windows programs, a short prolog of executable code precedes every exported far function in a module. Unless you program in assembly language, far-function prologs are generated by your high-level language compiler. For example, the Microsoft C compiler generates a prolog for every far function you compile using the */Gw* command-line switch.

The function prolog copies a data-segment value from register AX to DS, as shown in Figure 2-13. The correct data-segment value must be placed in AX by the function's caller before it transfers control to the function prolog.

```
nop
nop
nop
inc  bp         ; save BP+1 on stack
push bp
mov  bp,sp      ; save current stack pointer in BP
push ds         ; save current DS
mov  ds,ax      ; copy AX to DS
```

Figure 2-13.
Exported far-function prolog in a Windows application.

The data-segment value in AX is placed there in an instance thunk, a short piece of executable code that is created by calling the Windows API function *MakeProcInstance.* As shown in Figure 2-14, an instance thunk does nothing but store a data-segment value in AX and jump to the far-function prolog. By using a different instance thunk for each exported function in every application instance, Windows associates the correct default data segment with every far function without needing multiple copies of each function's executable code.

```
mov  ax,xxxx             ; store the data-segment value in AX
jmp  FAR PTR function    ; jump to exported FAR function
```

Figure 2-14.
An instance thunk. This piece of executable code is created by MakeProcInstance. *The instance thunk stores a data-segment value in register AX and jumps to the exported-function prolog shown in Figure 2-13.*

There are two ways in which you can disrupt this neat arrangement. One is simply to forget to export a far function that should have been exported. In this case, the should-have-been-exported function uses the calling function's default data segment instead of its own. This oversight causes problems as soon as the called function tries to access its static data or variables.

Suspect this bug whenever a program's static data seems to vanish. For example, the Microsoft C compiler stores string constants in a program's default data segment. Therefore, if you forget to export a function, all of the function's far pointers to the default data segment will be invalid. When this happens, string constants used as far-function parameters will seem to disappear. Consider the following C statement:

```
MessageBox( hWnd, "Hello, world", "This is a test", MB_OK );
```

The *MessageBox* function uses far string pointers (*LPSTR*) as parameters. If the function containing *MessageBox* should have been exported but was not, both of the string parameters will be invalid, and *MessageBox* will fail to display the proper text.

Another way that you might lose a function's data segment is if you forget to call *MakeProcInstance* to create an instance thunk for a properly exported far function. It is easy to forget to do this because you need not call *MakeProcInstance* for every exported far function. In particular, Windows itself creates an instance thunk when you call *RegisterClass* for an exported far function you use as a window function. However, you must call *MakeProcInstance* for all other exported far functions in an application. If you don't create an instance thunk for an exported far function that needs one, the data-segment value used by the function may not be valid. In that case, any references to data stored in the default data segment will be invalid. Because there is no way for a debugger to tell you that you should have created an instance thunk, consider whether you need to use *MakeProcInstance* whenever you suspect a dissociated data segment in an exported far function.

Invalid far pointers to moveable data

Some of the subtlest wild-pointer errors occur as a consequence of the normal operation of Windows' memory manager. As the memory manager allocates a new block of memory in the global heap or enlarges an existing memory block, it can rearrange the global heap by relocating other moveable blocks of global memory. This is normally not a problem in protected mode (standard or enhanced modes), but it can unmask wild pointers in real-mode applications.

The reason lies in the way a block of global memory is addressed in the different CPU modes. In standard and enhanced modes, a global-memory address is a 32-bit value that consists of a 16-bit selector and a 16-bit offset. The selector designates an entry in an 80286, 80386, or 80486 descriptor table. Part of each entry in the descriptor table is the address of the start of the memory block, which can lie anywhere in the CPU's address space. When Windows' memory manager moves a block of memory, it updates the block's starting address in the corresponding descriptor table. This means that the 32-bit selector:offset address you use in a Windows application remains unchanged, even if the memory manager moves a memory block to a different location in physical memory.

In real mode, however, there are no descriptor tables. A global-memory address consists of a 16-bit segment and a 16-bit offset. The segment value corresponds directly to a physical location in memory. If Windows' memory manager moves a

block of memory, the block's global-memory address contains a different segment value. Consequently, if an application stores a global-memory address in a pointer variable, the address will be invalidated if Windows' memory manager moves the global-memory block referenced by the pointer.

The reason this kind of bug is hard to find is that you cannot always control or determine when the memory manager will move a particular block of memory. The pointer may remain valid for a long time before the memory manager relocates the memory block to which it points. Moreover, when the memory manager moves a block of memory, it does not necessarily store new data at the block's previous address. This means that a wild pointer can continue to point to reasonable data even though it's pointing to the wrong location in memory.

The way to start looking for this kind of bug is to scrutinize your application's use of far pointers. Suspect any far pointers to data in the application's default data segment. Look carefully at the way the application accesses data in memory blocks allocated dynamically by *GlobalAlloc*.

What you are looking for is a far pointer that is used after any function call that might cause the global heap to be rearranged. Because it can be very hard to decide whether a given function call can lead to global-memory movement, the best way to ensure that a far pointer remains valid is to ensure that the pointer references a block with a fixed location in memory.

If the memory block is allocated by a call to *GlobalAlloc* that doesn't specify GMEM_FIXED in its first parameter, be sure that far pointers to the block are used only between a pair of calls to *GlobalLock* and *GlobalUnlock*, as shown in Figure 2-15. The call to *GlobalUnlock* should be close enough to the *GlobalLock* call that you can easily see them as a pair in your source code. Don't worry if you end up including a few extra calls to *GlobalLock* and *GlobalUnlock*. Eliminating the possibility of a wild pointer is well worth the minuscule overhead involved in locking and unlocking the memory block.

In real mode, Windows can also move the global-memory block that contains an application's default data segment. If you are chasing a wild pointer in a real-mode application, look for far pointers to data in the default data segment. You can avoid this kind of problem by using near pointers instead of far pointers to data in the default data segment and stack. Figures 2-16 and 2-17 illustrate the wrong and right ways to point to data in the default data segment.

```
BYTE GetOneByte( GLOBALHANDLE hData )
{
  LPSTR   lpData;

  /* lpData contains a valid far pointer */
  lpData = GlobalLock( hData );
  ⋮
  /* (use lpData here) */
  ⋮
  GlobalUnlock( hData );      /* lpData is no longer trustworthy */
  ⋮
  /* if global memory is moved here, lpData might be invalid */
  ⋮
  return *lpData;
}
```

Figure 2-15.
This function is supposed to return the first byte of data in a moveable global-memory block. However, if the block moves after the call to GlobalUnlock, *the far pointer* lpData *might become a wild pointer.*

```
void Create50EditWindows( HWND hWnd, LPSTR szText )
{
  int   i;

  for( i = 0; i < 50; i++ )
    CreateWindow( "Edit", szText, ... );
}
```

Figure 2-16.
The wrong way to point to a data item in an application's default data segment is to place a far pointer in a variable or in a function parameter. Here, one of the calls to CreateWindow *might cause Windows' memory manager to increase the size of the default data segment to accommodate the edit-control text. In real mode, the default data segment could move and invalidate* szText.

```
void Create50EditWindows( HWND hWnd, NPSTR szText )
{
  int   i;

  for( i = 0; i < 50; i++ )
    CreateWindow( "Edit", szText, ... );
}
```

Figure 2-17.
The right way to point to the data item is to use a near pointer. CreateWindow *requires a far pointer, so the C compiler generates code that converts* szText *into a far pointer dynamically each time* CreateWindow *is called.*

The function *Create50EditWindows* is called with the address of a buffer, *szText*, that is located in the default data segment. The only difference between the two versions of *Create50EditWindows* is the way the parameter *szText* is passed. In Figure 2-16, *szText* is a far pointer that can become invalid after *CreateWindow* executes. In Figure 2-17, *szText* is a near pointer that is converted to a far pointer dynamically each time *CreateWindow* is called.

When you suspect a wild pointer to a moveable memory block, try using fixed global-memory blocks instead of moveable blocks. Use GMEM_FIXED instead of GMEM_MOVEABLE as a parameter to *GlobalAlloc*, and use FIXED instead of MOVEABLE in your module-definition (.DEF) file to specify fixed segments. If the bug disappears, you're probably on the right track. In particular, if the bug disappears only when you fix a particular block of memory, look carefully at pointers to data in that memory block.

Trapping a wild pointer

If you think you have a wild pointer and you don't see a problem in your source code or module-definition file, you must trap the wild pointer by analyzing its bad effects. With luck, the wild pointer will point to the middle of a buffer whose contents you can examine. If you can identify the data inadvertently stored in the buffer when the wild pointer was used, you should be able to find the wild pointer in your source code.

When you can't find the wrongly stored data, you should take a more systematic approach. Start by running the application in protected mode (standard or enhanced mode). In protected mode, any of the following wild-pointer errors will lead to an "Unrecoverable Application Error" message:

- An attempt to store data in an executable-code segment.
- An attempt to use a far pointer with an invalid selector (such as might occur with an unitialized or null pointer).
- An attempt to read or write beyond the extent of a block of memory—that is, with a pointer whose selector is valid but whose offset points outside of the allocated size of the block.

These errors are normally trapped by the memory-protection feature of the 80286, 80386, and 80486 processors. To locate the pointer that causes one of these errors, run your application under CodeView in the debugging version of Windows. CodeView will trap the error and display this message:

```
Trap 13 (0DH) - General Protection Fault
```

Then CodeView will highlight the location in your application where the wild pointer is used.

Unfortunately, Windows version 3 does not make use of CPU memory protection to insulate an application's data segments from wild pointers in other applications. This means that a wild pointer in an application can cause corrupted data in a different application or even within Windows itself. If you suspect this kind of problem, the next step is to try to unmask the wild pointer by causing it to point to a protected memory location. One way to do this is to allocate some memory on the global heap before you run the application again. For example, you can run the Heap Walker utility, which is included in the Windows SDK, and use the commands in the Alloc menu to allocate a chunk out of the global heap. Then, when your application executes, its global-memory blocks will be allocated at different locations than they were before. With luck, the wild pointer will refer to a block of protected memory, and CPU memory protection will help you trap the bug.

Another technique is to run the same application in a different CPU mode. For example, if you're debugging an application in enhanced mode, try running it again either in standard mode or in real mode. This approach works because the selector or segment value of a far pointer depends on the CPU mode. Although a wild pointer might contain a valid selector value in one CPU mode, the value might be invalid in a different CPU mode and generate a memory-protection error.

You can trap a few wild pointers by running the debugging version of Windows in real mode. For example, you can sometimes cause a wild pointer to manifest itself by using the Shaker utility in the Windows SDK to shuffle the global heap. Shaker rapidly allocates and frees random blocks of global memory. This causes the memory manager to frequently relocate moveable global-memory blocks, so an application that contains a wild pointer fails sooner than it would without the Shaker utility.

Another real-mode technique is to use *ValidateCodeSegments* and *ValidateFreeSpaces*. *ValidateCodeSegments* causes Windows to compute a checksum on all global-memory blocks that contain executable code. If the checksum has changed, the debugging kernel issues fatal-exit error "Segment contents invalid" (error code 0x0409). To use *ValidateCodeSegments*, you must include the following statement in the *[kernel]* section of WIN.INI:

```
EnableSegmentChecksum=1
```

You can then insert calls to *ValidateCodeSegments* into strategic locations in your source code. If *ValidateCodeSegments* fails during your program's execution, you can track down the bug by inserting calls to *ValidateCodeSegments* until the wild pointer is isolated.

A related real-mode debugging technique is to use *ValidateFreeSpaces*. A call to *ValidateFreeSpaces* checks all free blocks in global memory for spurious data stored because of a wild pointer. As with *ValidateCodeSegments*, you should insert calls to *ValidateFreeSpaces* in your source code as frequently as you need to check for the use of a wild pointer.

To use *ValidateFreeSpaces*, you must include two statements in the *[kernel]* section of WIN.INI:

```
EnableFreeChecking=1
EnableHeapChecking=1
```

These statements cause Windows to store the value 0CCH (hexadecimal CC) in each byte of each unused global-memory block in the global heap. When you call *ValidateFreeSpaces*, Windows verifies that the free-memory blocks are still filled with 0CCH and issues a fatal-exit error if they are not. The value 0CCH is used because it represents a debugging breakpoint instruction in the 8086 family of CPUs. If you are using SYMDEB to debug a real-mode application and the application jumps to an invalid address in a block of memory that contains the value 0CCH, the resulting debugging break lets you examine the stack and trace backward to the function that jumped to the invalid address. In this situation, the invalid address is likely to have resulted from overwriting a return address on the stack.

Bulletproofing

A robust, industrial-strength Windows application must run properly in a variety of unfavorable situations. The best way to ensure that a Windows program can resist the slings and arrows of outrageous users is to test, redesign, and debug the application in the face of all the adverse circumstances you can anticipate.

CPU Modes

Before you distribute a Windows application to its users, be certain the application works properly in real, standard, and enhanced modes. If you developed the application in real mode, running it in standard and enhanced modes can catch memory-protection errors and wild pointers that might escape your attention in real mode. If you developed the program in standard or enhanced modes,

testing it in real mode can unmask errors such as wild pointers caused by the movement of global-memory blocks and failure to match calls to *GlobalLock* and *GlobalUnlock*.

It is also a good idea to validate your application by running it in real mode with LIM 4.0 expanded memory. This is a must if your application allocates shared global-memory blocks for DDE communications or if you are testing a dynamic link library that allocates global memory when called by an application. If an application works perfectly in the other CPU modes but fails in real mode with expanded memory, look for improper global-memory allocations, such as failing to use GMEM_DDESHARE or GMEM_NOT_BANKED as a parameter in the allocation of a shared global-memory block.

Input Overflow

A Windows application becomes susceptible to input overflow whenever a user can type or click the mouse button faster than the application can respond. Some Windows users are very fast typists. Others are very slow typists who unintentionally press keys so long that the keys autorepeat. Still others are mouse users who do not yet have the hand-to-eye coordination necessary to position the mouse and to click or double-click accurately. All these users can crash a Windows application that is not carefully designed to withstand input overflow.

For example, any Windows application that uses a push button to initiate a prolonged operation is at risk of input overflow because a user can inadvertently initiate the same operation several times by repeatedly clicking a mouse button or holding down the Enter key. To avoid this problem, call *EnableWindow* to disable the button when it is clicked and to reenable the button only after the prolonged operation has completed:

```
case WM_COMMAND:
  if( IDBUTTON == wParam )
  {
    EnableWindow( (HWND)LOWORD(lParam), FALSE );
    :
    /* carry out a prolonged operation */
     :
    EnableWindow( (HWND)LOWORD(lParam), TRUE );
```

Low-Memory Errors

Another way to bulletproof your application is to run it when global memory is scarce. When memory is at a premium, Windows' memory manager is much more likely to move and discard your application's moveable and discardable

global-memory blocks than it is when plenty of global memory is available. This means that bugs arising from the careless use of *GlobalAlloc* and *GlobalReAlloc* and wild pointers related to the movement of global-memory blocks are more likely to appear.

To verify that an application runs in a low-memory situation, run Heap Walker along with the application. Select Allocate All Of Memory in Heap Walker's Alloc menu. Then run the application and watch for bugs. The application will probably run very slowly as Windows' memory manager frequently moves or discards the application's code and data. If the application runs too slowly to be usable, you can speed it up a bit by using Heap Walker to free 25 or 50 KB.

Running the Wrong Way

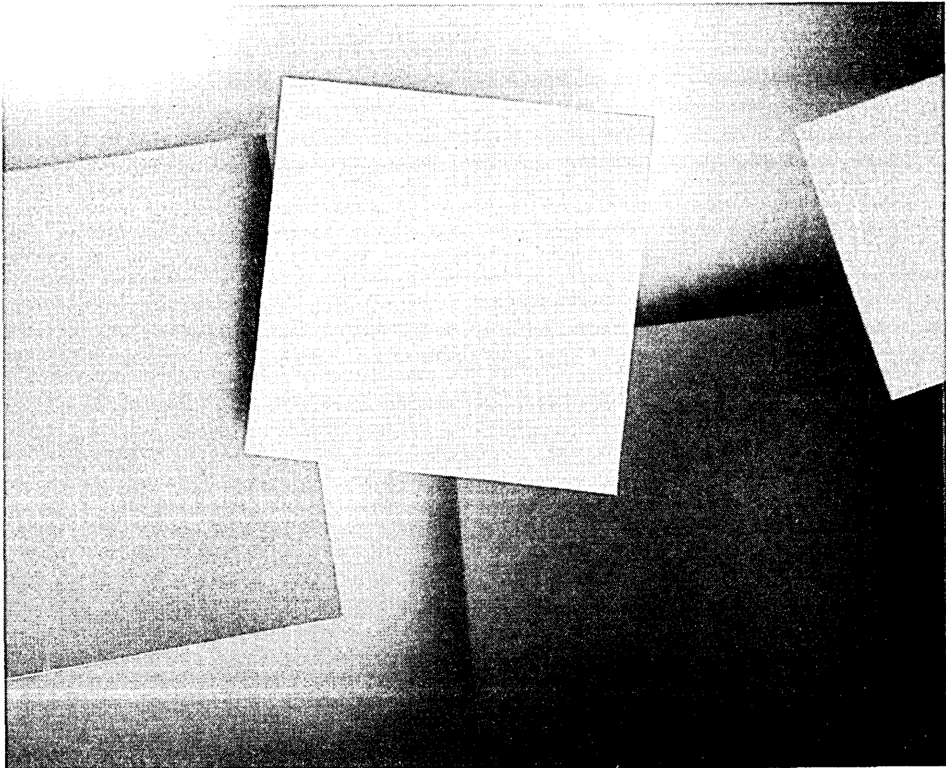
Before you let an unsuspecting user run a Windows application, try to anticipate how the user might make the application fail. This may not be easy if you're the one who designed the application because you naturally know the right way to run it. In this case, you might want to find a friend who will look at the application with a fresh viewpoint and ask some hard questions about what will happen when the application is run the wrong way.

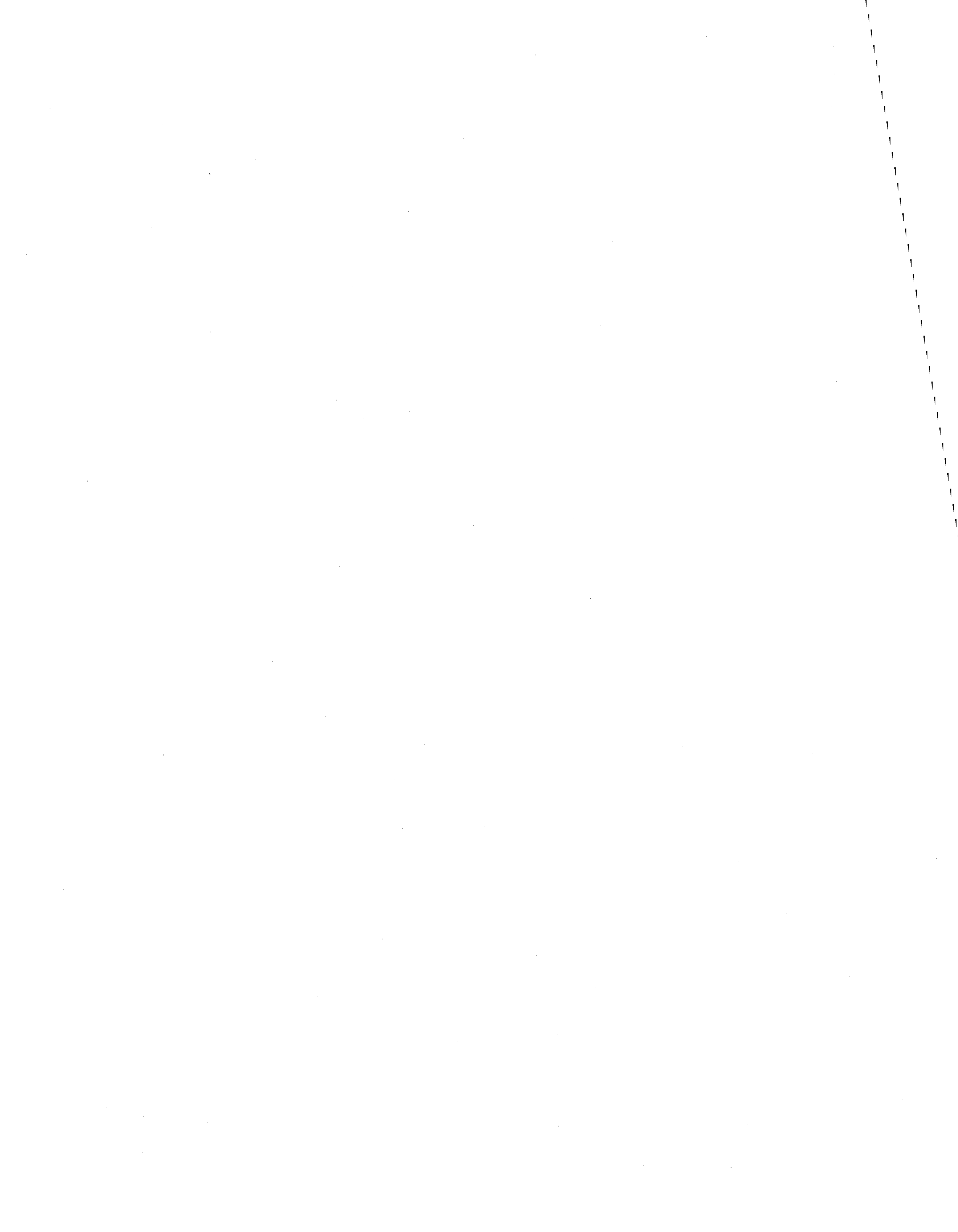
For example: Does everything still work properly when the user runs multiple instances of your application? Does the application work acceptably if the user stores data files on a floppy disk instead of on a hard disk? What happens if the user's hard disk runs out of space? When the user halts the application by ending the Windows session, does the application terminate gracefully, without leaving open files or GDI objects behind? Does it recover properly from hardware failures such as communication line drops, printer errors, and floppy-disk errors?

It may require some extra programming effort to be able to give good answers to questions like these, but bulletproofing will certainly save you trouble in the long run. The more experience you gain in debugging and careful bulletproofing in the Windows environment, the better your Windows applications will become.

3

Dynamic Link Libraries (DLLs)





The dynamic link library (DLL) is Windows' basic tool for sharing executable code and data. DLLs can be used to support shared subroutines, window classes, and read-only resources such as fonts, strings, and bitmaps. DLLs provide a way to extend and customize the Windows environment because the functions and resources you implement in DLLs can be used by applications in the same way as the ones predefined in the Windows API.

Like a stand-alone Windows application, a Windows dynamic link library is a Windows module that consists of executable code or data that Windows can load into memory. Functions defined in a DLL have full access to the Windows API. They can allocate memory, load resources, process input, and generate output. The important conceptual difference between a dynamic link library and a stand-alone application is that, unlike an application, a DLL is not a task. DLLs do not contain message-dispatching loops that call *GetMessage* or *DispatchMessage*. Instead, a DLL contains functions, fonts, bitmaps, or other resources that are called or loaded on demand by functions in other modules.

Structure of a Dynamic Link Library

The structure of a DLL resembles that of a stand-alone application. The difference is that a DLL has no *WinMain* function. Instead, DLLs contain an initialization function, *LibMain*, which Windows calls when it first loads the library into memory. DLLs must also contain an exit function, *WEP*, which Windows calls just before it discards the library from memory. The source code for DLLBASE.DLL, a baseline dynamic link library, shown in Figure 3-1, contains examples of both *LibMain* and *WEP*.

```
/****************************************************************************
*                                                                           *
* INIT.C                                                                    *
*                                                                           *
****************************************************************************/

#define NOCOMM
#include    <windows.h>

/*** GLOBAL VARIABLES ***/
HANDLE      hDLLInst;
```

Figure 3-1. *(continued)*
Source code for DLLBASE.DLL.

Figure 3-1. *continued*

```
/*** FUNCTION PROTOTYPES ***/
BOOL PASCAL FAR LibMain( HANDLE, WORD, WORD, LPSTR );


/************************************************************************
*                                                                      *
* LibMain                                                              *
*   Called by LibEntry when this DLL is loaded.                        *
*                                                                      *
************************************************************************/

BOOL PASCAL FAR
LibMain( HANDLE hInst, WORD wDS, WORD wHeapSize, LPSTR lpCmdTail )
{
    BOOL    bRVal = TRUE;


    /* if LibEntry has called LocalInit, unlock the default data segment */
    if( wHeapSize )
      bRVal = UnlockSegment( wDS );

    /* save the DLL instance handle in a global variable */
    hDLLInst = hInst;

    /* (other initialization would go here) */


    return bRVal;
}
```

```
/************************************************************************
*                                                                      *
* DLLBASE.C                                                            *
*   Simple Windows DLL.                                                *
*                                                                      *
* Exports:      Easter                                                 *
*               ShowDLLIcon                                            *
*                                                                      *
************************************************************************/

#define NOCOMM
#include    <windows.h>
```

(continued)

Figure 3-1. *continued*

```
/*** GLOBAL VARIABLES ***/
extern HANDLE   hDLLInst;                    /* defined in LIBMAIN.C */


/*** FUNCTION PROTOTYPES ***/
DWORD PASCAL FAR    Easter( int );
void PASCAL FAR     ShowDLLIcon( HWND, int, int, int );


/************************************************************************
 *                                                                      *
 * Easter                                                               *
 *   Returns the date of Easter for the specified year >= 1583.         *
 *   The return value is formatted as MAKELONG( day, month ).           *
 *                                                                      *
 *   Source:  Duffett-Smith, "Practical Astronomy with Your Calculator," *
 *            Copyright 1979, Cambridge University Press                *
 *                                                                      *
 ************************************************************************/

DWORD PASCAL FAR Easter( int y )
{
    int     a,b,c,d,e,f,g,h,i,k,l,m,n,p;


    a = y % 19;
    b = y / 100;
    c = y % 100;
    d = b / 4;
    e = b % 4;
    f = (b + 8) / 25;
    g = (b - f + 1) / 3;
    h = (19*a + b - d - g + 15) % 30;
    i = c / 4;
    k = c % 4;
    l = (32 + 2*e + 2*i - h - k) % 7;
    m = (a + 11*h + 22*l) / 451;
    n = (h + l - 7*m + 114) / 31;    /* 3=March; 4=April */
    p = (h + l - 7*m + 114) % 31;    /* day of month (0-30) */

    return MAKELONG(p+1, n);
}
```

(continued)

Figure 3-1. *continued*

```
/****************************************************************************
*                                                                          *
* ShowDLLIcon                                                              *
*   Displays the specified icon at the specified coordinates.              *
*                                                                          *
****************************************************************************/

void PASCAL FAR ShowDLLIcon( HDC hDC, int nID, int nX, int nY )
{
    HICON   hIcon;


    hIcon = LoadIcon( hDLLInst, MAKEINTRESOURCE(nID) );
    DrawIcon( hDC, nX, nY, hIcon );
    FreeResource( hIcon );
}
```

```
/****************************************************************************
*                                                                          *
* WEP.C                                                                    *
*   Windows exit procedure for Windows 3.x.                                *
*                                                                          *
* Exports: WEP RESIDENTNAME                                                *
*                                                                          *
* Notes:   This function must reside in a fixed code segment.              *
*                                                                          *
****************************************************************************/

#define NOCOMM
#include    <windows.h>


/****************************************************************************
*                                                                          *
* WEP                                                                      *
*   Called by Windows when this DLL is unloaded.                           *
*                                                                          *
****************************************************************************/

int PASCAL FAR WEP( int nParam )
{
    int     nRVal;
```

(continued)

Figure 3-1. *continued*

```
    switch( nParam )
    {
      case WEP_SYSTEM_EXIT:
      case WEP_FREE_DLL:
      default:
        nRVal = 1;
    }

    return nRVal;
}
```

```
/***************************************************************************
*                                                                         *
* DLLBASE.RC resource script                                              *
*                                                                         *
***************************************************************************/

STRINGTABLE
{
  100, "Wer zuletzt lacht lacht am Besten."
  101, "He who laughs last laughs best."
  102, "Rira bien qui rira le dernier."
}

100     ICON    deutsch.ico
101     ICON    usa.ico
102     ICON    france.ico
```

```
;***************************************************************************
;                                                                          *
;  DLLBASE.DEF module-definition file                                      *
;                                                                          *
;***************************************************************************

LIBRARY         DLLBASE
DESCRIPTION     'DLLBASE version 1.0'
EXETYPE         WINDOWS

CODE            LOADONCALL MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE SINGLE
```

(continued)

Figure 3-1. *continued*

```
HEAPSIZE       0

SEGMENTS       INIT_TEXT       PRELOAD DISCARDABLE
               WEP_TEXT        PRELOAD FIXED

EXPORTS        WEP             @1 RESIDENTNAME
               Easter          @2
               ShowDLLIcon     @3
```

Initialization Function

When Windows loads a library into memory, it transfers control to a short startup routine called *LibEntry*. This startup routine is defined in an object file named LIBENTRY.OBJ, which Microsoft supplies as part of the Windows SDK. (You must link LIBENTRY.OBJ into your DLL on the LINK command line.) *LibEntry* performs two actions. If you declare a nonzero heap size in the library's module-definition file, *LibEntry* calls *LocalInit* to initialize a local heap in the DLL's default data segment. Then *LibEntry* transfers control to the library's initialization function, *LibMain*.

Because *LibMain* always executes at the time Windows loads the library, the *LibMain* function can carry out any initialization required by other functions in the library. You can use *LibMain* to initialize global variables, register window classes, or perform any other initialization needed by the library. *LibMain* returns a nonzero value to indicate that initialization is successful. If *LibMain* returns 0, Windows does not load the library.

Typically, *LibMain* also calls *UnlockData* or *UnlockSegment* for libraries that use a local heap. This is necessary because *LibEntry* calls *LocalInit* if the library contains a local heap, and *LocalInit* leaves the DLL's default data segment locked. For a DLL that might be used when Windows is running in real mode, it is best to leave the default data segment unlocked so that Windows' memory manager can move it if necessary. In the sample library, DLLBASE, *LibMain* determines whether to call *UnlockSegment* by testing the value of the *wHeapSize* parameter.

Library Functions

If a function defined in a DLL is to be called by another Windows module, the function must be exported so that Windows can dynamically link the caller with the called function. Such functions in DLLs must also be defined as far functions

because the library code segment in which the function is defined might not be the same as the segment from which the function will be called. In DLLBASE, the application-callable functions are *Easter* and *ShowDLLIcon*, both of which are declared with the keywords PASCAL FAR and exported in the library's module-definition file, DLLBASE.DEF.

Although DLLBASE defines only two application-callable functions, you can incorporate as many functions as you want into a library as long as you export each application-callable function. The only functions that do not need to be exported are those that are called only by other functions in the same library. However, there is no penalty for exporting such functions. In a DLL, exported functions can be called by other functions in the same DLL as well as by functions in other modules.

Library Resources

DLLBASE includes a loadable STRINGTABLE resource and three ICON resources, defined in the resource file DLLBASE.RC. The resources are built into the DLLBASE.DLL file, just as they would be in a Windows application. When Windows loads DLLBASE.DLL, the resources become available to any application. To access the STRINGTABLE resource, for example, an application must first call *LoadLibrary* to ensure that Windows has loaded the library and to obtain the library's module handle. The application then uses the handle in calling *LoadString* to access the string table. A function in the same DLL accesses a DLL resource by using the DLL's instance handle instead of calling *LoadLibrary* to obtain a module handle. The function *ShowDLLIcon* in DLLBASE.C shows how to do this.

The Exit Procedure

Every DLL must export a callback function named *WEP* (which stands for Windows Exit Procedure). Windows calls a library's *WEP* function while it is unloading the library from memory. The purpose of the exit procedure is to provide a way for a DLL to perform any final actions that need to be carried out before the library is discarded. Such actions might include freeing any memory blocks that have been allocated in the library, discarding strings and bitmaps, or closing open files.

There are two important restrictions on what a *WEP* function can do: A *WEP* function must not call *LoadLibrary* or *FreeLibrary*, and *WEP* should not call a function in another DLL if the other DLL is being unloaded at the same time. For example, imagine that you define a function named *MyPrintf* in DLL1.DLL and

that you import the *MyPrintf* function into a second DLL using an import library or an IMPORTS statement in the second DLL's .DEF file:

```
IMPORTS     DLL1.MyPrintf
```

In this situation, it would be an error for the second DLL's *WEP* function to call *MyPrintf*:

```
int PASCAL FAR WEP( int nParam )
{
  /* WRONG */
  MyPrintf( "The second DLL is unloading" );

  return 1;
}
```

Windows passes a parameter to *WEP* that indicates the circumstance under which the library is being unloaded. If the Windows session is terminating, the parameter has the value WEP_SYSTEM_EXIT. If all applications that use the library have terminated or have freed the library, the parameter's value is WEP_FREE_DLL. In all cases, the *WEP* function should return the value 1.

The Library Reference Count

For each DLL in memory, Windows maintains a reference count that indicates the number of tasks that are dynamically linked to the library. Windows increments the reference count when it loads an instance of an application that calls a library function and decrements the count when the instance terminates. The reference count is also incremented by the *LoadLibrary* function and decremented by the *FreeLibrary* function. A DLL can examine its own reference count by calling *GetModuleUsage* with the library's module handle as a parameter.

Managing Segments

If you have never used a DLL, you might wonder why it is necessary to take the trouble to build a DLL just to share functions or data. Why not build a stand-alone Windows application that contains the executable code for the functions you want to share? You could export each of the shared functions with an appropriate EXPORTS statement in the application's module-definition (.DEF) file. Then, in any application that called one of the shareable functions, you would include a corresponding IMPORTS statement in the .DEF file.

This technique actually works, but it is very unreliable. One obvious problem is that it is hard to guarantee that the application that contains the shared functions

will be loaded in memory at the moment another application attempts to call them. Another problem is more subtle: When expanded memory is available, Windows' memory manager uses bank-switched expanded memory to contain an application's code segments. When the application is not actively processing a Windows message, the expanded-memory banks containing the code segments are not mapped into the CPU address space. Any shared functions in a banked-out code segment would be inaccessible to other applications.

Dynamic link libraries were designed to avoid these problems. After a DLL is loaded into memory, it remains loaded until it is released by every application that uses it. Also, when an application calls a DLL function, Windows' memory manager ensures that the executable code segment that contains the function is always accessible, even if the memory manager has discarded the segment or swapped it into a bank of expanded memory.

In this way, Windows' memory manager takes care of the memory-management problems involved with sharing library functions. Your job is to structure your DLL's code and data segments so as to use memory efficiently.

Memory Models for DLLs

For a small DLL, it is simplest to use the small memory model so that the library is loaded into memory in one code segment and one default data segment. For a large DLL, however, you can optimize memory usage by compiling the DLL with a medium memory model and dividing the DLL's source code into two or more separately compiled segments.

When you use the medium memory model, each segment of executable code should be no larger than about 4 KB. This is the size of the virtual memory block that Windows' memory manager uses when Windows executes in enhanced mode on an 80386 or 80486 microprocessor. When you link a medium memory model DLL, mark its code segments as moveable with the following CODE statement in the module-definition file:

```
CODE        MOVEABLE
```

This statement marks all code segments in the DLL as moveable. You should override this statement by using a SEGMENTS statement for the following segments:

- The code segment that contains the *WEP* function.
- Code segments that contain interrupt handlers.

- Code segments that contain the callback function specified in a call to *GlobalNotify*.

These code segments should be declared as fixed rather than moveable. This is illustrated in the module-definition file DLLBASE.DEF in Figure 3-1.

You can further tune a DLL's memory management in the module-definition file by using the PRELOAD and DISCARDABLE attributes for INIT_TEXT, the code segment that contains the library startup function *LibEntry*. The PRELOAD attribute makes sense because this segment contains code that executes when the library is first loaded. The DISCARDABLE attribute is used because the segment will not be used again as long as the library remains in memory. The same reasoning applies to the segment that contains *LibMain*, so you might want to compile *LibMain* along with *LibEntry* in the discardable INIT_TEXT segment. In the DLLBASE example, this is accomplished by defining *LibMain* in a file named INIT.C so that the Microsoft C compiler names the corresponding segment INIT_TEXT. You could achieve the same result using the compiler's */NT* (name code segment) command-line switch.

If a DLL references large amounts of static data, you might consider using a large memory model. However, far data segments in the large memory model are fixed segments. It is better to use a small or medium memory model and load static data as a binary resource. Avoiding fixed data segments makes Windows' virtual memory management more efficient.

The Default Data Segment

Unlike stand-alone applications, Windows libraries cannot have multiple instances—that is, a Windows library has only one default data segment. The data in a library's default data segment is shared by all applications that use the library. Because only one copy of a library's data segment can exist, you must use the SINGLE attribute with the DATA statement in a library's .DEF file.

With DLLs, you have another option. If a library uses no static data and calls no functions that need to use the library's local heap, you can avoid using a data segment at all. To do this, enter this statement in your module-definition file:

```
DATA    NONE
```

You can also avoid using a data segment by adding the keyword NODATA to each EXPORTS statement. If you create a library without a data segment, be very careful how you use data in library functions. The following C-language function

contains several examples of what you cannot do in a library without a default data segment; the comments explain why not:

```
int  nBytes;                    /* external variables are stored
                                   in the default data segment */
char szName[] = "Alpha";        /* initialized variables are stored
                                   in the default data segment */


int MyFunc()
{
  static LOCALHANDLE hX;        /* static variables are stored
                                   in the default data segment */

  lstrcmp( szName, "Beta" );    /* "Beta" is implicitly stored
                                   in the default data segment */
}
```

There is yet another memory-management optimization that involves the DLL's default data segment. A DLL's initialization function, *LibMain*, should usually contain a call to *UnlockData* or *UnlockSegment*:

```
UnlockSegment( wDS );
```

This call ensures that the library's default data segment remains unlocked while the library remains in memory. Although this may not matter when Windows is run in protected mode, it can improve memory management in a real-mode Windows environment. In real mode, unlocking the data segment lets Windows' memory manager move the segment to a different location in the global heap in response to other modules' demands for global memory.

If you are designing a DLL for real-mode execution and you are using a far pointer to an item in the default data segment, you might want to lock the data segment temporarily to ensure that the far pointer remains valid while it is being used. You can do this by surrounding the code that uses the far pointer with a pair of calls to *LockData* and *UnlockData* or *LockSegment* and *UnlockSegment*:

```
static char Buffer[64];        /* (stored in the DLL's
                                  default data segment) */


LockSegment( wDS );            /* lock the default data segment */
MyFunction( (LPSTR)&Buffer );  /* use a far pointer */
UnlockSegment( wDS );          /* unlock the data segment */
```

Calling Library Functions

Shared functions in dynamic link libraries must be exported and must be called with far calls. In this regard, they are just like exported functions in stand-alone Windows applications. Nevertheless, there are differences in the methods that exported library functions use to manage the default data segment and the stack.

Far-Function Prologs

Windows associates a different default data segment with each DLL and each instance of a task. This is accomplished by a short prolog of executable code that precedes every exported far function. The function prolog sets up the CPU's DS register so that the function uses the proper default data segment when it references static data and data on the stack.

Every far function in a Windows module is preceded by a function prolog. You need not code the prolog explicitly—it is generated by your compiler or, if you program in assembly language, by a macro expansion. (The Microsoft C compiler's */Gw* switch instructs the compiler to generate Windows function prologs.) The prolog generated by the compiler, which is shown in Figure 3-2, does not change the default data-segment value in register DS. However, if a far function is exported, Windows modifies the executable code in the function prolog at the time it loads the function into memory so that the prolog stores the appropriate data-segment value in DS when it executes.

```
push ds          ; copy DS to AX
pop  ax
nop
inc  bp          ; save BP+1 on stack
push bp
mov  bp,sp       ; save current stack pointer in BP
push ds          ; save current DS
mov  ds,ax       ; copy AX to DS
```

Figure 3-2.
Nonexported far-function prolog. This prolog does not change the default data-segment value in register DS.

Windows modifies exported far-function prologs differently in applications and in DLLs. In an application, the prolog copies a default data-segment value from register AX to register DS, as in Figure 3-3. The data-segment value, which corresponds to the instance of the application in which the function is executing, is

```
nop
nop
nop
inc  bp        ; save BP+1 on stack
push bp
mov  bp,sp     ; save current stack pointer in BP
push ds        ; save current DS
mov  ds,ax     ; copy AX to DS
```

Figure 3-3.
Exported far-function prolog in a Windows application. This prolog changes the default data segment by copying the value in register AX to register DS.

placed in AX in a fragment of executable code called an instance thunk (shown in Figure 3-4). For each exported far function, an application must create an instance thunk by calling *MakeProcInstance*, unless the function is a window function whose address was passed as a parameter to *RegisterClass*.

```
mov  ax,xxxx           ; store the data-segment value in AX
jmp far ptr function   ; jump to exported far-function prolog
```

Figure 3-4.
An instance thunk. This piece of executable code is created by MakeProcInstance. *The value stored in AX is used in the exported-function prolog shown in Figure 3-3.*

A different arrangement is used for exported far functions in a DLL. There is only one data segment associated with a dynamic link library, so there is no need for Windows to support multiple instance thunks, and there also is no need to call *MakeProcInstance* for exported library functions. In a DLL, the data-segment value is built into the prolog of each far function, as in Figure 3-5. The structural differences between the different far-function prologs are worth remembering when you debug a program that contains a wild pointer caused by an improperly exported far function. In this situation, you can verify that a far function is correctly exported by examining the function's prolog.

```
mov  ax,xxxx   ; store the data-segment value in AX
inc  bp        ; save BP+1 on stack
push bp
mov  bp,sp     ; save current stack pointer in BP
push ds        ; save current DS
mov  ds,ax     ; copy AX to DS
```

Figure 3-5.
Exported far-function prolog in a dynamic link library. This prolog stores the library's default data-segment value in register DS.

Parameter-Passing Conventions

When you pass parameters between dynamic link libraries and other Windows modules, keep in mind that the library's code and data segments are not the same as those in other modules. All pointers to library data items and all calls to library functions must use far pointers so that they explicitly specify the segment to which they refer.

The order in which you specify parameters is not constrained by anything in Windows as long as the caller and the function being called agree on the parameter order. The Windows API generally uses the Pascal parameter-passing convention, which is that the leftmost parameter is the first one pushed on the stack. You can, however, use the C-language convention, which is that the rightmost parameter is the first one pushed on the stack.

The advantage of the Pascal convention is that the executable code that manages the stack across function calls is a few bytes shorter and a little faster than the equivalent code required for the C method. The advantage of the C-language convention is that it lets you design functions such as *printf* that support a variable number of parameters.

Unless you need a variable-length parameter list, you should use the Pascal convention. If you use the C convention, remember that the C compiler adds an underscore character to the beginning of each function name, so EXPORTS module-definition statements for such function names must include the underscore. For example, consider the following C function:

```
int far cdecl MyPrintf();
```

The corresponding EXPORTS statement in the module-definition file would contain an underscore:

```
EXPORTS      _MyPrintf
```

The Stack and the Default Data Segment

In all Windows programs, the stack is where function parameters are passed, where a function's return address is saved when it calls another function, and where storage for automatic variables is allocated. The default data segment is used for static data, for constants, and for the local heap. In Windows applications, which use the small or medium memory model, these two logically distinct types of data are maintained in a single segment—that is, the stack is located in the default data segment. In this way, stack data and static data can be addressed by using offsets within the same segment.

This is a handy arrangement because a program can store the same value in the CPU's DS (data segment) register and SS (stack segment) register and then use either register to address both the stack and the rest of the data in the default data segment. This makes it easier for a compiler such as the Microsoft C compiler to translate source code into executable code because the compiler can generate code that uses the DS and SS registers interchangeably to address both static data (stored in the default data segment) and automatic data (stored on the stack).

A DLL differs from a Windows application in that a DLL does not have its own stack. Instead, each function in a DLL uses its caller's stack. This can lead to addressing problems because it contradicts the compiler's assumption that the DS and SS registers contain the same segment value. When you write DLL source code, you must keep this potential problem in mind.

In general, the way to avoid DLL addressing problems is to use far pointers to DLL data items instead of near pointers. If you use a near pointer, the compiler might not be able to determine whether the pointer refers to a location in the stack or a location in the default data segment. In this case, a compiler might arbitrarily assume that the reference is to the default data segment instead of the stack. This is not a valid assumption in a DLL, where the stack segment is not the same as the default data segment.

In contrast, when you use a far pointer, you refer explicitly to a particular segment as well as to the offset of a data item within the segment. When a compiler compiles a far pointer, it makes no assumptions about segment addressing, and you make no assumptions about the compiler's assumptions. This is the strategy adopted by the designers of the Windows API, in which functions almost always expect far pointers as parameters.

The problem with using far pointers everywhere in a DLL is that DLLs are typically compiled with the small or medium memory models, and small-model and medium-model C runtime library functions expect near pointers as parameters. You can sometimes solve this problem by using a far-pointer equivalent of a library function. For example, the *_fmemmove* function in the Microsoft C libraries is a far-pointer equivalent of the *memmove* function.

If you do use near pointers, be sure they always refer to data items in the DLL's data segment and not to a variable or an array on the stack. This means that near pointers should refer only to static variables and arrays and to memory blocks allocated in the DLL's local heap with *LocalAlloc*. The Microsoft C compiler can help you to avoid inaccurate use of near pointers. When you use the */Aw* command-line switch, the compiler will warn you if you use a near pointer to a data item on the stack.

Figure 3-6 is an example of how to use a far pointer to access data on the stack. The function *AuxPrintf* uses *OutputDebugString* to send a formatted text string to the debugging display.

```
void FAR cdecl AuxPrintf( LPSTR szFmt, ... )
{
  char         szBuf[128];
  LPSTR FAR *  pArg1;


  /* point to the second parameter on the stack */
  pArg1 = ((LPSTR FAR *)&szFmt) + 1;

  /* format the output string */
  wvsprintf( szBuf, szFmt, (LPSTR)pArg1 );

  /* display the formatted string */
  OutputDebugString( szBuf );
}
```

Figure 3-6.
Using a far pointer to data on the stack. The address of szFmt *is cast to a far pointer when it is assigned to the variable* pArg1.

The formatting is carried out through a call to *wvsprintf*, which expects a pointer to an argument list as a parameter. *AuxPrintf* stores the pointer as a far pointer:

```
pArg1 = ((LPSTR FAR *)&szFmt) + 1;
```

The far pointer to *szFmt* is stored correctly as a reference to the stack segment. Compare this to a similar statement that stores a near pointer instead of a far pointer:

```
pArg1 = ((LPSTR *)&szFmt) + 1;      /* WRONG! */
```

The difference between these two statements becomes apparent in the call to *wvsprintf*:

```
wvsprintf( szBuf, szFmt, (LPSTR)pArg1 );
```

In the first example, the value of *pArg1* is passed to *wvsprintf* as a far pointer. This works perfectly. In the second example, however, the C compiler converts the near pointer to a far pointer by incorrectly assuming that the pointer lies in the default data segment. The point is clear: Use far pointers to address stack variables in Windows DLLs.

Sharing Functions and Data

It is not hard to share functions or data in a DLL. Often all you need to use are exported far functions and far data pointers. However, Windows provides important alternatives to the use of far pointers—namely, global-memory handles and resources. The best method for sharing a particular function or data item depends on whether the item lies outside the DLL or within it.

Pointers to Data Outside a DLL

If you keep in mind how a DLL uses its default data segment and stack, you can use far data pointers to pass data to DLL functions. There are situations, however, where relying on a far pointer can lead to problems.

One such situation occurs when Windows executes in real mode. In real mode, a far pointer to a block of memory consists of a physical-segment address plus an offset within the segment. The physical-segment value can become invalid if Windows' memory manager moves the block of memory to which the pointer refers. To avoid this problem, be sure that far pointers always refer to memory blocks whose location in memory is fixed.

The easiest way to fix the location of a block of memory is either to use the FIXED keyword to declare a fixed segment in a module-definition file or to use the GMEM_FIXED flag in a call to *GlobalAlloc*. The problem with doing this is that such segments remain at the same physical location in memory from the time they are allocated until the time they are freed. This limits the memory manager's ability to rearrange the global heap dynamically in response to the memory requirements of multiple applications.

A better approach is to use the keyword MOVEABLE in the module-definition file and the GMEM_MOVEABLE flag in *GlobalAlloc*. To pass a far pointer to data in a moveable memory block, you must first call *GlobalLock* to lock the block's global-memory handle:

```
lpDATA = GlobalLock( hData );
MyDLLFunc( lpData );         /* call a DLL function named
                                  MyDLLFunc */
GlobalUnlock( hData );
```

To pass a far pointer to data in an application's default data segment, call *LockSegment(-1)* or *LockData* prior to calling the DLL function:

```
LockData();
MyDLLFunc( (LPSTR)&Data );    /* call a DLL function named
                                  MyDLLFunc */
UnlockData();
```

A simpler approach to passing data from an application to a DLL is to use a handle to a global-memory block as a parameter instead of a far pointer. With this technique, the DLL function itself can lock the memory handle:

```
void PASCAL FAR MyDLLFunction ( GLOBALHANDLE hData )
{
  LPSTR lpData;

  lpData = GlobalLock( hData );
         :
  /* use lpData */
         :
  GlobalUnlock( hData );
}
```

Dereferencing the global-memory handle involves a bit of extra programming—the DLL must call *GlobalLock* and *GlobalUnlock* each time it accesses data passed to it from an application. However, using a global handle is a safer technique than using a far data pointer because you can control how Windows' memory manager manages the memory block that contains the data item. This is most important in real mode, where far pointers can be invalidated by the normal operation of Windows' memory manager. In particular, you can use the GMEM_DISCARDABLE or GMEM_NOT_BANKED flags with *GlobalAlloc* to specify whether the memory manager is allowed to discard a particular memory block or store it in a bank of expanded memory.

Use *GlobalAlloc* with the GMEM_DDESHARE flag to ensure that a global-memory block remains accessible within a DLL when Windows is using real-mode expanded memory. Consider, for example, what might happen if a far pointer referred to data that was owned by an application executing in bank-switched expanded memory. If the application called a DLL function that in turn caused a different application to execute, the calling application might be banked out of memory and the far pointer would no longer be valid. Careful use of GMEM_NOT_BANKED will avoid this problem.

Pointers to Data Within a DLL

When you pass data from a DLL to an application, you must be certain that far pointers to such data remain valid while the application is using them. Again, this is most important in real mode. For example, you can pass far pointers to shared data stored within a DLL's default data segment as long as you ensure that Windows' memory manager will not move the data segment while the pointer is in use.

Again, a more general approach to sharing DLL data is for a DLL to allocate blocks of global memory to contain shared data. The DLL can then use global handles instead of far pointers to refer to the shared data. Although this technique involves the extra overhead of allocating a block of global memory and of dereferencing a global-memory handle, it decreases the possibility of creating wild pointers when Windows is running in real mode.

Pointers to Functions Outside a DLL

To pass the address of a function located outside a DLL to a DLL function, you must use a far pointer to the function or to the function's instance thunk. If the far function is also in a DLL, you need only pass the address of the function:

```
MyDLLFunc( (FARPROC)DialogFunc );
```

If the far function is defined in an application, you must pass the address of an instance thunk for the function:

```
pThunk = MakeProcInstance( (FARPROC)DialogFunc, hInstance );
MyDLLFunc( pThunk );
FreeProcInstance( pThunk );
```

Pointers to Functions Within a DLL

To pass the address of a library function to an application, use a far pointer to the function. The application can subsequently call the function by dereferencing the far pointer, as in Figure 3-7. Of course, a library need not contain a function that returns pointers to other library functions—an application can call *GetProcAddress* to obtain a pointer to any exported DLL function. In either case, be sure you export any application-callable library functions with appropriate EXPORTS statements in the library's module-definition file.

```
/* in the DLL */
FARPROC PASCAL FAR NthDLLFunction( int n )
{
  switch( n )
  {
    case 1:
      return Function1;
      break;
```

Figure 3-7. *(continued)*
Returning a DLL function pointer to an application. All of the DLL functions are exported in the DLL's module-definition file.

Figure 3-7. *continued*

```
      case 2:
        return Function2;
        break;

      default:
        return DefaultFunction;
        break;
    }


}

void PASCAL FAR Function1( ... )
{
    :
}

void PASCAL FAR Function2( ... )
{
    :
}

void PASCAL FAR DefaultFunction( ... )
{
    :
}
```

```
/* in the calling application */
FARPROC    lpDLLFn;


/* get a pointer to the first DLL function */
lpDLLFn = NthDLLFunction( 1 );

/* execute the DLL function */
(*lpDLLFn)(...);
```

Using Resources

Another way to use a DLL to share data is to define resources within the library. An application can access a resource in a library by calling *LoadLibrary* followed by the appropriate load function (*LoadBitmap*, *LoadIcon*, *LoadString*, and so on). For example, Figure 3-8 shows how an application would access one of the string resources defined in the sample library DLLBASE.DLL.

```
HANDLE    hLibrary;
char      szBuf[64];


hLibrary = LoadLibrary( "DLLBASE.DLL" );
if( 32 >= hLibrary )
{
  LoadString( hLibrary, 101, szBuf, sizeof szBuf );
  FreeLibrary( hLibrary );
}
```

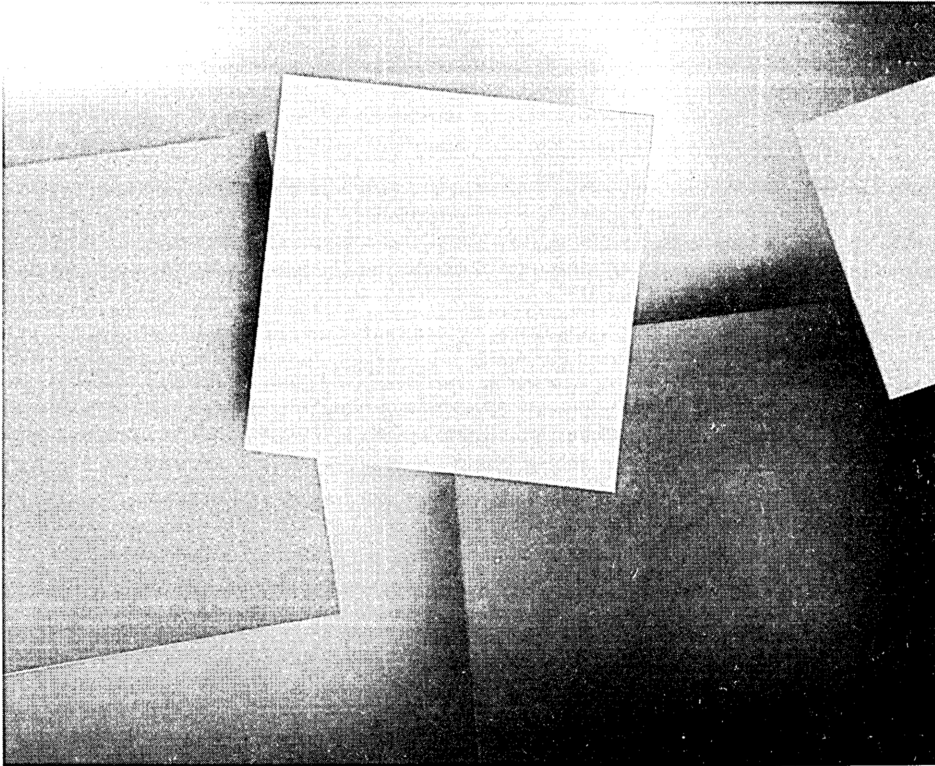
Figure 3-8.
Using a resource defined in a dynamic link library. The value 101 refers to the identifier used in the string table resource in DLLBASE.RC. (See Figure 3-1 on page 61.)

You can define any resource in a library that you can define in a stand-alone application, including GDI objects, dialog boxes, menus, and user-defined binary data. You might even want to create a library that contains only resources. The only exported function in such a library would be its *WEP* function. Such resource-only libraries provide an elegant mechanism for sharing globally used objects in a portable, low-overhead manner.

Many libraries are designed as a combination of shared resources and useful functions. One specific application of this kind of library is to support a window class that describes a user-defined custom control. In fact, custom-control libraries are important enough in the Windows environment to be the subject of the next chapter.

4

Custom Controls



Controls are the building blocks of a Windows application's visual interface. The predefined control classes—Static, Button, Edit, ScrollBar, ListBox, and ComboBox—are general-purpose tools that you can use in a wide variety of programming situations. It is hard to imagine a Windows program that does not somewhere use at least one button, scroll bar, or static text control.

It is not hard, however, to implement custom controls whose function or appearance is tailored to the specific needs of an application. Like the predefined controls, custom controls are child windows that perform specific visual input or output functions. You can develop a custom control as part of a Windows application, but you can make a custom-control window class available to multiple applications by implementing it in a dynamic link library and by making the control accessible to resource editors such as the Dialog Editor (DIALOG.EXE) in the Windows SDK.

A Custom Control in a Windows Application

The simplest way to implement a custom control is to have an application call *RegisterClass* to register the custom-control class and then call *CreateWindow* to create one or more control windows. You can use the same methods to test and debug the control that you use to test and debug any stand-alone Windows application. You can even add CONTROL statements to DIALOG resources in the application's resource script to create custom controls in dialog boxes, provided that the application registers the control class before it loads the dialog resources that use the custom control.

Figure 4-1 on the following page shows a single example of a custom-control class named *RYG*, whose visual appearance is a familiar red-yellow-green symbol. A program can set or get the state of an RYG control by sending it one of two user-defined messages, RYG_SETSTATE or RYG_GETSTATE. The source code in Figure 4-2 on the following pages shows how the RYG class is developed within a Windows application.

In this example, the application is nothing more than a testbed for developing the *RYG* control class. Embedding the control in an application makes it easy to test and modify the control's user interface. When the control class works properly, you can use it in a different application simply by copying the relevant source code.

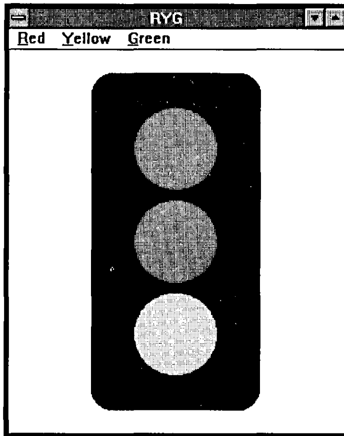


Figure 4-1.
A typical RYG *control displayed by RYGDEV.EXE.*

```
#**************************************************************************
#                                                                         *
# NMAKE description for RYGDEV.EXE                                        *
#                                                                         *
#**************************************************************************

.c.obj:
    cl /AM /c /G2sw /Osw /W4 /Zlp $*.c

ALL:            rygdev.exe

ryg.obj:        ryg.c rygdev.h

rygdev.obj:     rygdev.c rygdev.h

rygdev.res:     rygdev.rc rygdev.h rygdev.ico
    rc /r rygdev.rc

rygdev.exe:     ryg.obj rygdev.obj rygdev.res rygdev.def
    link /al:16 /nod /noe rygdev ryg, , , libw mlibcew, rygdev.def
    rc rygdev.res
```

Figure 4-2. *(continued)*
Source code for RYGDEV.EXE.

Figure 4-2. *continued*

```
/*********************************************************************
*                                                                    *
* RYGDEV.C                                                           *
*    Simple application that uses the RYG control from RYG.C         *
*                                                                    *
* Exports:      TopLevelWndFn                                        *
*                                                                    *
*********************************************************************/

#define NOCOMM
#include    <windows.h>
#include    "rygdev.h"

#define IDRYG   100

/*** FUNCTION PROTOTYPES ***/

LONG PASCAL FAR TopLevelWndFn( HWND, WORD, WORD, LONG );

static HWND     Init( HANDLE, HANDLE, int );

/*** GLOBAL VARIABLES ***/

HBRUSH  hRYGBrush[3];
DWORD   dwRGB[3] = { RGB(0xFF,0x00,0x00),               /* red */
                     RGB(0xFF,0xFF,0x00),               /* yellow */
                     RGB(0x00,0xFF,0x00) };             /* green */

static char     szTopLevelClass[] = "RYG:TopLevel";
static char     szRYGClass[] = "RYG";
static char     szAppTitle[] = "RYG";


/*********************************************************************
*                                                                    *
* WinMain                                                            *
*                                                                    *
*********************************************************************/

int PASCAL
WinMain( HANDLE hInst, HANDLE hPrevInst, LPSTR lpszCmdLine, int nCmdShow )
{
    HWND    hWnd;
    MSG     msg;
    int     n;
```

(continued)

Figure 4-2. *continued*

```
    hWnd = Init( hInst, hPrevInst, nCmdShow );
    if( !hWnd )
      return 0;

    /* create brushes */
    for( n=0; n<3; n++ )
      hRYGBrush[n] = CreateSolidBrush( dwRGB[n] );

    while( GetMessage( &msg, 0, 0, 0 ) )
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }

    /* destroy brushes */
    for( n=0; n<3; n++ )
      DeleteObject( hRYGBrush[n] );

    return msg.wParam;
}


/*************************************************************************
*                                                                        *
* Init                                                                   *
*                                                                        *
*************************************************************************/

static HWND Init( HANDLE hInst, HANDLE hPrevInst, int nCmdShow )
{
    WNDCLASS    wc;
    HWND        hWnd;

    if( !hPrevInst )
    {
      /* register the top-level window class */
      wc.lpszClassName    = szTopLevelClass;
      wc.hInstance        = hInst;
      wc.lpfnWndProc      = TopLevelWndFn;
      wc.hCursor          = LoadCursor( 0, IDC_ARROW );
      wc.hIcon            = LoadIcon( hInst, "TopLevelIcon" );
      wc.lpszMenuName     = "TopLevelMenu";
      wc.hbrBackground    = COLOR_WINDOW+1;
      wc.style            = CS_HREDRAW | CS_VREDRAW;
      wc.cbClsExtra       = 0;
      wc.cbWndExtra       = 0;
```

(continued)

Figure 4-2. *continued*

```
        if( !RegisterClass( &wc ) )
          return 0;                      /* return 0 if unsuccessful */

        /* register the RYG window class */
        wc.lpszClassName    = szRYGClass;
        wc.hInstance        = hInst;
        wc.lpfnWndProc      = RYGWndFn;
        wc.hCursor          = LoadCursor( 0, IDC_ARROW );
        wc.hIcon            = 0;
        wc.lpszMenuName     = NULL;
        wc.hbrBackground    = COLOR_WINDOW+1;
        wc.style            = 0;
        wc.cbClsExtra       = 0;
        wc.cbWndExtra       = sizeof(WORD);

        if( !RegisterClass( &wc ) )
          return 0;                      /* return 0 if unsuccessful */
      }

      /* create and display a top-level window */
      hWnd = CreateWindow( szTopLevelClass,
                           szAppTitle,
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                           0,
                           0,
                           hInst,
                           NULL );

      /* create an RYG window */
      CreateWindow( szRYGClass,
                           "",
                           WS_CHILD | WS_VISIBLE,
                           0, 0, 16, 16,
                           hWnd,
                           IDRYG,
                           hInst,
                           NULL );

      ShowWindow( hWnd, nCmdShow );
      UpdateWindow( hWnd );

      return hWnd;
    }
```

(continued)

Figure 4-2. *continued*

```
/*******************************************************************
 *                                                                 *
 * TopLevelWndFn                                                   *
 *                                                                 *
 *******************************************************************/

LONG PASCAL FAR
TopLevelWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bDWP = FALSE;

    switch( wMsg )
    {
      case WM_COMMAND:
        SendDlgItemMessage( hWnd, IDRYG, RYG_SETSTATE, wParam, 0L );
        break;

      case WM_SIZE:
        MoveWindow( GetDlgItem( hWnd, IDRYG ),
                        0, 0, LOWORD(lParam), HIWORD(lParam), TRUE );
        break;

      case WM_DESTROY:
        PostQuitMessage( 0 );
        break;

      default:
        bDWP = TRUE;
        break;
    }

    if( bDWP )
      lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

    return lRVal;
}
```

```
/*******************************************************************
 *                                                                 *
 * RYG.C                                                           *
 *    RYG custom-control implementation                            *
 *                                                                 *
 * Exports:      RYGWndFn                                          *
 *                                                                 *
 *******************************************************************/
```

(continued)

Figure 4-2. *continued*

```
#define NOCOMM
#include    <windows.h>
#include    "rygdev.h"

/*** FUNCTION PROTOTYPES ***/

static void     RYGPaint( HWND );
static void     RYGShowState( HDC, int, BOOL );
static void     RYGSetMapMode( HWND, HDC );

/*** GLOBAL VARIABLES ***/

extern HBRUSH   hRYGBrush[];                    /* defined in RYGDEV.C */
extern DWORD    dwRGB[];

static int  nY[3] = { 800, 250, -300 };


/************************************************************************
 *                                                                      *
 * RYGWndFn                                                             *
 *                                                                      *
 ************************************************************************/

LONG PASCAL FAR RYGWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    HDC     hDC;
    LONG    lRVal = 0L;
    BOOL    bDWP = FALSE;

    switch( wMsg )
    {
      case WM_PAINT:
        RYGPaint( hWnd );
        break;

      case RYG_SETSTATE:
        hDC = GetDC( hWnd );
        RYGSetMapMode( hWnd, hDC );
        RYGShowState( hDC, SetWindowWord( hWnd, 0, wParam ), FALSE );
        RYGShowState( hDC, wParam, TRUE );
        ReleaseDC( hWnd, hDC );
        break;

      case RYG_GETSTATE:
        lRVal = (DWORD)GetWindowWord( hWnd, 0 );
        break;
```

(continued)

Figure 4-2. *continued*

```
        case WM_CREATE:
          SetWindowWord( hWnd, 0, -1 );
          break;

        default:
          bDWP = TRUE;
          break;
      }

    if( bDWP )
      lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

    return lRVal;
}


/*************************************************************************
 *                                                                       *
 * RYGPaint                                                              *
 *                                                                       *
 *************************************************************************/

static void RYGPaint( HWND hWnd )
{
    HDC         hDC;
    PAINTSTRUCT ps;
    HBRUSH      hBrush;
    int         n, nState;

    hDC = BeginPaint( hWnd, &ps );
    RYGSetMapMode( hWnd, hDC );

    /* background */
    hBrush = SelectObject( hDC, GetStockObject( DKGRAY_BRUSH ) );
    RoundRect( hDC, -500, 1000, 500, -1000, 250, 250 );
    SelectObject( hDC, hBrush );

    /* dots */
    nState = GetWindowWord( hWnd, 0 );
    for( n=0; n<3; n++ )
      RYGShowState( hDC, n, (n == nState) );

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

(continued)

Figure 4-2. *continued*

```
/**************************************************************************
*                                                                         *
* RYGShowState                                                            *
*                                                                         *
**************************************************************************/

static void RYGShowState( HDC hDC, int nState, BOOL bColor )
{
    HBRUSH  hBrush;

    if( (nState < 0) || (nState >= 3) )
      return;

    if( bColor )
      hBrush = SelectObject( hDC, hRYGBrush[nState] );
    else
      hBrush = SelectObject( hDC, GetStockObject( GRAY_BRUSH ) );

    Ellipse( hDC, -250, nY[nState], 250, nY[nState]-500 );
    SelectObject( hDC, hBrush );
}

/**************************************************************************
*                                                                         *
* RYGSetMapMode                                                           *
*                                                                         *
**************************************************************************/

static void RYGSetMapMode( HWND hWnd, HDC hDC )
{
    RECT    rect;

    /* set up an isotropic coordinate system centered in the client area */
    GetClientRect( hWnd, &rect );
    SetMapMode( hDC, MM_ISOTROPIC );
    SetWindowExt( hDC, 1000, 1000 );
    SetViewportExt( hDC, rect.right/2, -rect.bottom/2 );
    SetViewportOrg( hDC, rect.right/2, rect.bottom/2 );
}
```

(continued)

Figure 4-2. *continued*

```
/**************************************************************************
*                                                                         *
* RYGDEV.RC resource script                                               *
*                                                                         *
**************************************************************************/

#include    <windows.h>
#include    "rygdev.h"

/* icons */
TopLevelIcon  ICON    rygdev.ico

/* menus */
TopLevelMenu    MENU
{
  MENUITEM    "&Red"    RYG_RED
  MENUITEM    "&Yellow" RYG_YELLOW
  MENUITEM    "&Green"  RYG_GREEN
}
```

```
/**************************************************************************
*                                                                         *
* RYGDEV.H                                                                *
*                                                                         *
**************************************************************************/

#define RYG_SETSTATE    (WM_USER+0)
#define RYG_GETSTATE    (WM_USER+1)

#define RYG_RED         0
#define RYG_YELLOW      1
#define RYG_GREEN       2

/*** FUNCTION PROTOTYPES ***/

/* defined in RYG.C */
LONG PASCAL FAR RYGWndFn( HWND, WORD, WORD, LONG );
```

(continued)

Figure 4-2. *continued*

```
;****************************************************************************
;                                                                           *
; RYGDEV.DEF module-definition file                                         *
;                                                                           *
;****************************************************************************

NAME            RYGDEV
DESCRIPTION     'RYGDEV.EXE version 1.0'
EXETYPE         WINDOWS
STUB            'WINSTUB.EXE'

CODE            LOADONCALL MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE MULTIPLE

SEGMENTS        _TEXT   PRELOAD MOVEABLE DISCARDABLE

HEAPSIZE        1024
STACKSIZE       5120

EXPORTS         TopLevelWndFn
                RYGWndFn
```

A Custom Control in a Dynamic Link Library

The problem with building a custom control directly into an application is that it is hard to use the same control in two different applications at the same time. You can't simply register the same control class in two different applications because a class definition remains valid only until you terminate the last instance of the application that registered the class. A second application that uses the class will crash if the first application terminates and the control's window function disappears. If you implement a custom-control class in an application and other applications use the class, you must devise a method to ensure that the application is executing, that it has registered the custom-control class, and that it will continue to execute until all other applications have finished using the custom control.

You might accomplish this through the devious use of API functions such as *FindWindow*, *WinExec*, and *UnregisterClass*, but Windows offers a more straightforward alternative: the dynamic link library. It is slightly more difficult to implement a custom control in a DLL than in a stand-alone application, but the effort is worthwhile because custom controls in DLLs can be used in multiple applications, including resource editors such as the SDK Dialog Editor.

To illustrate this, compare the source code for RYG.DLL in Figure 4-3 with the source code for the application in Figure 4-2. Much of the source code in the application was copied directly into the DLL's source code. Actions that are carried out during initialization, such as class registration and creation of GDI objects, appear in *LibMain* in the DLL. Similarly, the DLL's *WEP* function supports actions that are performed when the control class is no longer needed. The source code in RYG.C, which embodies the functionality of the control class, is used in the DLL just as it is in the stand-alone application.

```
#*************************************************************************
#                                                                        *
# NMAKE description for RYG.DLL                                          *
#                                                                        *
#*************************************************************************

.c.obj:
    cl /AMw /c /D _WINDOWS /D _WINDLL /G2sw /Osw /W4 /Zlp $*.c

ALL:            ryg.dll

init.obj:       init.c ryg.h

ryg.obj:        ryg.c ryg.h

wep.obj:        wep.c

ryg.dll:   ryg.obj init.obj wep.obj ryg.def
    link /al:16 /nod /noe libentry init ryg wep, ryg.dll, , \
      libw mdllcew, ryg.def
    rc ryg.dll
```

```
/*************************************************************************
 *                                                                       *
 * INIT.C                                                                *
 *   Initialization code for RYG.DLL                                     *
 *                                                                       *
 *************************************************************************/

#define NOCOMM
#include    <windows.h>
#include    "ryg.h"
```

Figure 4-3.
Source code for RYG.DLL.

(continued)

Figure 4-3. *continued*

```
/*** GLOBAL VARIABLES ***/

HANDLE  hDLLInst;
HBRUSH  hRYGBrush[3];
DWORD   dwRGB[3] = { RGB(0xFF,0x00,0x00),              /* red */
                     RGB(0xFF,0xFF,0x00),              /* yellow */
                     RGB(0x00,0xFF,0x00) };            /* green */

static char     szRYGClass[] = "RYG";

/*** FUNCTION PROTOTYPES ***/

BOOL PASCAL FAR     LibMain( HANDLE, WORD, WORD, LPSTR );

static BOOL         RegisterRYGClass( void );

/************************************************************************
*                                                                       *
* LibMain                                                               *
*                                                                       *
************************************************************************/

BOOL PASCAL FAR
LibMain( HANDLE hInst, WORD wDS, WORD wHeapSize, LPSTR lpCmdTail )
{
    BOOL    bRVal;
    int     n;

    /* if LibEntry has called LocalInit, unlock the default data segment */
    if( wHeapSize )
      UnlockSegment( wDS );

    /* save the DLL instance handle in a global variable */
    hDLLInst = hInst;

    /* register the custom-control class */
    bRVal = RegisterRYGClass();

    /* create brushes */
    if( bRVal )
      for( n=0; n<3; n++ )
        hRYGBrush[n] = CreateSolidBrush( dwRGB[n] );

    return bRVal;
}
```

(continued)

Figure 4-3. *continued*

```
/****************************************************************************
 *                                                                          *
 * RegisterRYGClass                                                         *
 *                                                                          *
 ****************************************************************************/

static BOOL RegisterRYGClass()
{
    WNDCLASS    wc;

    wc.lpszClassName    = szRYGClass;
    wc.hInstance        = hDLLInst;
    wc.lpfnWndProc      = RYGWndFn;
    wc.hCursor          = LoadCursor( 0, IDC_ARROW );
    wc.hIcon            = 0;
    wc.lpszMenuName     = NULL;
    wc.hbrBackground    = COLOR_WINDOW+1;
    wc.style            = CS_GLOBALCLASS;
    wc.cbClsExtra       = 0;
    wc.cbWndExtra       = sizeof(WORD);

    return RegisterClass( &wc );
}
```

```
/****************************************************************************
 *                                                                          *
 * RYG.C                                                                    *
 *   RYG custom-control implementation                                      *
 *                                                                          *
 * Exports:       RYGWndFn                                                  *
 *                                                                          *
 ****************************************************************************/

#define NOCOMM
#include    <windows.h>
#include    "ryg.h"

/*** FUNCTION PROTOTYPES ***/

static void     RYGPaint( HWND );
static void     RYGShowState( HDC, int, BOOL );
static void     RYGSetMapMode( HWND, HDC );
```

(continued)

Figure 4-3. *continued*

```
/*** GLOBAL VARIABLES ***/

extern HBRUSH   hRYGBrush[];                    /* defined in INIT.C */
extern DWORD    dwRGB[];
static int  nY[3] = { 800, 250, -300 };

/*************************************************************************
 *                                                                       *
 * RYGWndFn                                                              *
 *                                                                       *
 *************************************************************************/

LONG PASCAL FAR RYGWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    HDC      hDC;
    LONG     lRVal = 0L;
    BOOL     bDWP = FALSE;

    switch( wMsg )
    {
      case WM_PAINT:
        RYGPaint( hWnd );
        break;

      case RYG_SETSTATE:
        hDC = GetDC( hWnd );
        RYGSetMapMode( hWnd, hDC );
        RYGShowState( hDC, SetWindowWord( hWnd, 0, wParam ), FALSE );
        RYGShowState( hDC, wParam, TRUE );
        ReleaseDC( hWnd, hDC );
        break;

      case RYG_GETSTATE:
        lRVal = (DWORD)GetWindowWord( hWnd, 0 );
        break;

      case WM_CREATE:
        SetWindowWord( hWnd, 0, -1 );
        break;

      default:
        bDWP = TRUE;
        break;
    }

    if( bDWP )
      lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

    return lRVal;
}
```

(continued)

Figure 4-3. *continued*

```
/*********************************************************************
 *                                                                   *
 * RYGPaint                                                          *
 *                                                                   *
 *********************************************************************/

static void RYGPaint( HWND hWnd )
{
    HDC         hDC;
    PAINTSTRUCT ps;
    HBRUSH      hBrush;
    int         n, nState;

    hDC = BeginPaint( hWnd, &ps );
    RYGSetMapMode( hWnd, hDC );

    /* background */
    hBrush = SelectObject( hDC, GetStockObject( DKGRAY_BRUSH ) );
    RoundRect( hDC, -500, 1000, 500, -1000, 250, 250 );
    SelectObject( hDC, hBrush );

    /* dots */
    nState = GetWindowWord( hWnd, 0 );
    for( n=0; n<3; n++ )
      RYGShowState( hDC, n, (n == nState) );

    EndPaint( hWnd, &ps );
}

/*********************************************************************
 *                                                                   *
 * RYGShowState                                                      *
 *                                                                   *
 *********************************************************************/

static void RYGShowState( HDC hDC, int nState, BOOL bColor )
{
    HBRUSH  hBrush;

    if( (nState < 0) || (nState >= 3) )
      return;

    if( bColor )
      hBrush = SelectObject( hDC, hRYGBrush[nState] );
    else
      hBrush = SelectObject( hDC, GetStockObject( GRAY_BRUSH ) );

    Ellipse( hDC, -250, nY[nState], 250, nY[nState]-500 );
    SelectObject( hDC, hBrush );
}
```

(continued)

Figure 4-3. *continued*

```
/***********************************************************************
*                                                                      *
* RYGSetMapMode                                                        *
*                                                                      *
***********************************************************************/

static void RYGSetMapMode( HWND hWnd, HDC hDC )
{
    RECT    rect;

    /* set up an isotropic coordinate system centered in the client area */
    GetClientRect( hWnd, &rect );
    SetMapMode( hDC, MM_ISOTROPIC );
    SetWindowExt( hDC, 1000, 1000 );
    SetViewportExt( hDC, rect.right/2, -rect.bottom/2 );
    SetViewportOrg( hDC, rect.right/2, rect.bottom/2 );
}
```

```
/***********************************************************************
*                                                                      *
* WEP.C                                                                *
*   Windows exit procedure for RYG.DLL                                 *
*                                                                      *
* Exports:      WEP RESIDENTNAME                                       *
*                                                                      *
***********************************************************************/

#define NOCOMM
#include    <windows.h>

/*** GLOBAL VARIABLES ***/

extern HBRUSH   hRYGBrush[];                    /* defined in RYG.C */

/***********************************************************************
*                                                                      *
* WEP                                                                  *
*                                                                      *
***********************************************************************/

int PASCAL FAR WEP( int nParam )
{
    int     n;
```

(continued)

Figure 4-3. *continued*

```
    /* destroy brushes */
    for( n=0; n<3; n++ )
      DeleteObject( hRYGBrush[n] );

    return 1;
}
```

```
/************************************************************************
*                                                                       *
* RYG.H                                                                 *
*   Header file for RYG.DLL                                             *
*                                                                       *
************************************************************************/

#define RYG_SETSTATE     (WM_USER+0)
#define RYG_GETSTATE     (WM_USER+1)

#define RYG_RED          0
#define RYG_YELLOW       1
#define RYG_GREEN        2

/*** FUNCTION PROTOTYPES ***/

/* defined in RYG.C */
LONG PASCAL FAR RYGWndFn( HWND, WORD, WORD, LONG );
```

```
;***********************************************************************
;                                                                      *
; RYG.DEF module-definition file                                       *
;                                                                      *
;***********************************************************************

LIBRARY          RYG
DESCRIPTION      'RYG version 1.0'
STUB             'WINSTUB.EXE'
EXETYPE          WINDOWS

CODE             LOADONCALL MOVEABLE DISCARDABLE
DATA             PRELOAD MOVEABLE SINGLE

HEAPSIZE         0
```

(continued)

Figure 4-3. *continued*

```
SEGMENTS        INIT_TEXT       PRELOAD DISCARDABLE
                WEP_TEXT        PRELOAD FIXED

EXPORTS         WEP         @1  RESIDENTNAME
                RYGWndFn
```

Although RYG.DLL is small, it contains all four of the essential elements of a dynamic link library that supports a custom-control class:

- An initialization function (*LibMain*)
- A call to *RegisterClass* to register the control class
- A control-class window function
- An exit procedure (*WEP*)

Each of these components of the DLL's structure corresponds to a part of the application in which the custom-control class was originally developed.

DLL Initialization and Class Registration

A DLL's initialization function, *LibMain*, is the best place to call *RegisterClass* for a custom-control class. When you do this, you can be certain that *RegisterClass* will be called exactly once when Windows loads the library. In a DLL, a custom-control class should be registered with the CS_GLOBALCLASS style. This style makes the class registration available to all applications, not just the application that first loads the DLL.

The Control Class Window Function

It goes without saying that the window function that supports a custom-control class is always part of a custom-control DLL. If you develop a custom control by embedding it in an application, you can often transplant the window-function source code directly into the DLL.

The Windows Exit Procedure (*WEP*)

You should implement and export the *WEP* function in a custom-control library just as you would in any other DLL. Windows calls the *WEP* function when the DLL is no longer to be used, before it discards the library from memory.

The purpose of including *WEP* in a library is to provide a consistent method for library routines to clean up at the time the library is unloaded. In the case of

RYG.DLL, the *WEP* routine deletes the GDI objects that the *RYG* class uses to paint the control. You might also think that *WEP* would be a reasonable place to call *UnregisterClass* to unregister the custom-control class, but don't do it. Windows implicitly unregisters all classes that have been registered in a module at the time the last instance of the module is unloaded. Calling *UnregisterClass* in a DLL's *WEP* function is redundant.

Using a Custom Control

As it stands, RYG.DLL supports a usable custom-control class named *RYG*. To create *RYG* controls, an application first loads RYG.DLL. It can do this explicitly by calling *LoadLibrary* in its *WinMain* function:

```
hLibrary = LoadLibrary( "RYG.DLL" );
```

An application can also load the library implicitly by importing a library function. A simple way to do this is to include an appropriate IMPORTS statement in the application's module-definition file:

```
IMPORTS     RYG.WEP
```

After the library has been loaded, the application can create *RYG* controls directly by calling *CreateWindow* or indirectly through CONTROL statements in DIALOG resources in its .RC file.

The most important drawback to RYG.DLL is that a resource editor such as the SDK Dialog Editor, DIALOG.EXE, does not recognize the *RYG* control class. This makes it harder than it ought to be to design an application that uses *RYG* controls. You need to do some extra work to make a custom-control class known to a resource editor, but the payoff is a custom-control class that can be used within the editor just like one of the predefined Windows control classes.

Custom Controls and the Dialog Editor

The Windows SDK documentation describes the three interface functions that a resource editor can call to determine the characteristics of a custom-control class in a DLL. You should build these functions into a custom-control DLL to give resource editors such as the SDK Dialog Editor the ability to manipulate the custom control. (Other resource editors might use a different method to communicate with a custom-control DLL. See the documentation for your resource editor for details.)

The three resource-editor interface functions described in the Windows SDK are listed in Figure 4-4. The *Info* function lets a resource editor determine which custom-control styles are available and which style to use by default. The *Style* function provides a way for a resource editor to invoke a dialog box through which the user can specify the style of a particular custom control. The *Flags* function associates a string of symbolic names with a particular control's style.

Name	***Export Ordinal***	***Notes***
Info	2	Returns descriptive data about the control library to a resource editor in a CTLINFO data structure.
Style	3	Updates a CTLSTYLE data structure containing a style description for a particular control.
Flags	4	Builds a string of symbolic style names for inclusion in a CONTROL statement in a .RC or .DLG file.

Figure 4-4.
Resource-editor interface functions in a custom-control DLL.

A resource editor identifies the interface functions in a custom-control library by reference to the export ordinal numbers you assign. Although you must use the predefined ordinal values, you can choose any convenient names for these functions. It's good practice, though, to use names derived from the name of your custom control.

Because this interface relies on ordinal numbers to identify the resource-editor entry points in a library, you can define entry points for only one custom-control class per DLL. If you want to use several different custom controls in a resource editor, you may need to define each custom-control class in a separate DLL. The alternative is to define different controls as different styles in the same control class. The predefined static and button classes adopt this approach—the predefined static class includes, among others, static text, rectangle, and frame styles; the button class includes push buttons, check boxes, radio buttons, and group boxes.

A typical DLL interface to the Windows SDK Dialog Editor is illustrated in the source code in Figure 4-5 on the following page, which implements a custom-control library named COLORCTL.DLL. The custom-control class supported in COLORCTL.DLL is named *ColorCtl.* The purpose of the *ColorCtl* control is to fill its window rectangle with color. You set the control's color by specifying an RGB

value as the control's window text. The control's style determines whether the displayed shape is a rectangle, an ellipse, or a round-cornered rectangle and whether the control is displayed with a border. Several examples of *ColorCtl* controls are shown in Figure 4-6 on page 125.

```
#***********************************************************************
#                                                                      *
# NMAKE description for COLORCTL.DLL                                   *
#                                                                      *
#***********************************************************************

.c.obj:
    cl /AMw /c /D _WINDOWS /D _WINDLL /G2sw /Osw /W4 /Zlp $*.c

ALL:            colorctl.dll

colorctl.obj:   colorctl.c colorctl.h

dlgedit.obj:    dlgedit.c colorctl.h

init.obj:       init.c colorctl.h

wep.obj:        wep.c

colorctl.res:   colorctl.rc colorctl.h
    rc /r colorctl.rc

colorctl.dll:   colorctl.obj dlgedit.obj init.obj wep.obj \
                  colorctl.res colorctl.def
    link /al:16 /nod /noe libentry init colorctl dlgedit wep, \
      colorctl.dll, , libw mdllcew, colorctl.def
    rc colorctl.res colorctl.dll
```

```
/***********************************************************************
*                                                                      *
* INIT.C                                                               *
*    Initialization for COLORCTL.DLL                                   *
*                                                                      *
***********************************************************************/

#define NOCOMM
#include    <windows.h>
#include    "colorctl.h"
```

Figure 4-5. *(continued)*
Source code for COLORCTL.DLL.

Figure 4-5. *continued*

```
/*** GLOBAL VARIABLES ***/

HANDLE          hDLLInst;

static char     szColorCtlClass[] = COLORCTLCLASSNAME;

/*** FUNCTION PROTOTYPES ***/

BOOL PASCAL FAR     LibMain( HANDLE, WORD, WORD, LPSTR );

static BOOL         RegisterColorCtlClass( void );

/************************************************************************
 *                                                                      *
 * LibMain                                                              *
 *                                                                      *
 ************************************************************************/

BOOL PASCAL FAR
LibMain( HANDLE hInst, WORD wDS, WORD wHeapSize, LPSTR lpCmdTail )
{
    /* if LibEntry has called LocalInit, unlock the default data segment */
    if( wHeapSize )
    UnlockSegment( wDS );

    /* save the DLL instance handle in a global variable */
    hDLLInst = hInst;

    /* register the custom-control class */
    return RegisterColorCtlClass();
}

/************************************************************************
 *                                                                      *
 * RegisterColorCtlClass                                                *
 *                                                                      *
 ************************************************************************/

static BOOL RegisterColorCtlClass()
{
    WNDCLASS    wc;

    wc.lpszClassName    = szColorCtlClass;
    wc.hInstance        = hDLLInst;
    wc.lpfnWndProc      = ColorCtlWndFn;
    wc.hCursor          = LoadCursor( 0, IDC_ARROW );
    wc.hIcon            = 0;
    wc.lpszMenuName     = NULL;
```

(continued)

Figure 4-5. *continued*

```
    wc.hbrBackground    = 0;
    wc.style            = CS_DBLCLKS | CS_GLOBALCLASS;
    wc.cbClsExtra       = 0;
    wc.cbWndExtra       = 0;

    return RegisterClass( &wc );
}
```

```
/*************************************************************************
*                                                                        *
* COLORCTL.C                                                             *
*   Implementation of the ColorCtl control                               *
*                                                                        *
* Exports:      ColorCtlWndFn                                            *
*                                                                        *
*************************************************************************/

#define NOCOMM
#include    <windows.h>
#include    <stdlib.h>      /* strtoul */
#include    "colorctl.h"

/*** FUNCTION PROTOTYPES ***/

static void MsgPaint( HWND );
static BOOL MsgEraseBkgnd( HWND, HDC );

/*************************************************************************
*                                                                        *
* ColorCtlWndFn                                                          *
*                                                                        *
*************************************************************************/

LONG PASCAL FAR
ColorCtlWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bDWP = FALSE;

    switch( wMsg )
    {
      case WM_PAINT:
        MsgPaint( hWnd );
        break;
```

(continued)

Figure 4-5. *continued*

```
        case WM_NCPAINT:               /* don't paint the default border */
        case WM_NCCALCSIZE:
          break;

        case WM_ERASEBKGND:
          lRVal = MsgEraseBkgnd( hWnd, wParam );
          break;

        default:
          bDWP = TRUE;
          break;
      }

      if( bDWP )
        lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

      return lRVal;
  }

  /************************************************************************
  *                                                                       *
  * MsgPaint                                                              *
  *                                                                       *
  ************************************************************************/

  static void MsgPaint( HWND hWnd )
  {
      DWORD       dwStyle;
      DWORD       dwRGB;
      char        szRGB[12];
      HDC         hDC;
      PAINTSTRUCT ps;
      RECT        rect;
      HPEN        hPen;
      HBRUSH      hBrush;
      int         nCorner;

      hDC = BeginPaint( hWnd, &ps );

      /* get the style flags and color */
      dwStyle = GetWindowLong( hWnd, GWL_STYLE );
      GetWindowText( hWnd, szRGB, sizeof szRGB );
      dwRGB = HexToDWord( szRGB );

      /* get the client-area rectangle */
      GetClientRect( hWnd, &rect );
```

(continued)

Figure 4-5. *continued*

```
    /* select pen and brush */
    hBrush = CreateSolidBrush( dwRGB );
    hBrush = SelectObject( hDC, hBrush );

    if( dwStyle & WS_BORDER )
      hPen = GetStockObject( BLACK_PEN );
    else
    {
      hPen = GetStockObject( NULL_PEN );
      InflateRect( &rect,
                   GetSystemMetrics( SM_CXBORDER ),
                   GetSystemMetrics( SM_CYBORDER ) );
    }
    hPen = SelectObject( hDC, hPen );

    /* the control's style determines its shape */
    switch( dwStyle & (CCS_RECT | CCS_ROUND) )
    {
      case CCS_ROUND:
        Ellipse( hDC, 0, 0, rect.right, rect.bottom );
        break;

      case (CCS_RECT | CCS_ROUND):
        nCorner = MulDiv( min( rect.right, rect.bottom ), 2, 3 );
        RoundRect( hDC, 0, 0, rect.right, rect.bottom, nCorner, nCorner );
        break;

      case CCS_RECT:
      default:
        Rectangle( hDC, rect.left, rect.top, rect.right, rect.bottom );
        break;
    }

    hPen = SelectObject( hDC, hPen );
    hBrush = SelectObject( hDC, hBrush );
    DeleteObject( hBrush );

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

(continued)

Figure 4-5. *continued*

```
/***********************************************************************
*                                                                      *
* MsgEraseBkgnd                                                        *
*                                                                      *
***********************************************************************/
static BOOL MsgEraseBkgnd( HWND hWnd, HDC hDC )
{
    HBRUSH  hBrush;
    RECT    rect;

    hBrush = (HBRUSH)SendMessage( GetParent( hWnd ),
                                  WM_CTLCOLOR,
                                  hDC,
                                  MAKELONG(hWnd, CTLCOLOR_STATIC) );
    GetClientRect( hWnd, &rect );
    FillRect( hDC, &rect, hBrush );

    return TRUE;
}

/***********************************************************************
*                                                                      *
* HexToDWord                                                           *
*   Converts a hexadecimal ASCIIZ string to a DWORD.                   *
*                                                                      *
***********************************************************************/

DWORD PASCAL FAR HexToDWord( LPSTR pHex )
{
    LOCALHANDLE     hBuf;
    NPSTR           pBuf;
    static NPSTR    pEnd;
    DWORD           dwRVal;

    if( (NULL != pHex) && *pHex )
    {
      /* allocate a temporary buffer in the DLL's local heap */
      hBuf = LocalAlloc( LHND, lstrlen( pHex ) + 1 );
      pBuf = LocalLock( hBuf );
      lstrcpy( pBuf, pHex );

      /* convert the string */
      dwRVal = strtoul( pBuf, &pEnd, 16 );
```

(continued)

Figure 4-5. *continued*

```
        /* if non-hexadecimal string, return a default RGB value */
        if( *pEnd )
          dwRVal = DEFRGB;

        /* free the buffer */
        LocalUnlock( hBuf );
        LocalFree( hBuf );
      }

      else
        dwRVal = DEFRGB;

      return dwRVal;
  }
```

```
/*************************************************************************
 *                                                                       *
 * DLGEDIT.C                                                             *
 *   Entry points for Windows dialog editor for the ColorCtl control     *
 *                                                                       *
 * Exports:      ColorCtlInfo                                            *
 *               ColorCtlStyle                                           *
 *               ColorCtlFlags                                           *
 *               ColorCtlDlgFn                                           *
 *                                                                       *
 *************************************************************************/

#define NOCOMM
#include    <windows.h>
#include    <custcntl.h>
#include    <string.h>      /* _fmemcpy */
#include    "colorctl.h"

/*** GLOBAL VARIABLES ***/

extern HANDLE   hDLLInst;                   /* defined in INIT.C */

static LPFNSTRTOID      pStrToIdFn;
static LPFNIDTOSTR      pIdToStrFn;

static CTLINFO  CtlInfo =
                {
                 0x0100,                        /* wVersion */
                 3,                             /* wCtlTypes */
```

(continued)

Figure 4-5. *continued*

```
                   COLORCTLCLASSNAME,            /* szClass */
                   "",                           /* szTitle */
                   "",                           /* szReserved */

                   0, 40, 20, CCS_RECT | WS_CHILD | WS_VISIBLE,
                     "ColorCtl (rect)",
                   0, 30, 30, CCS_ROUND | WS_CHILD | WS_VISIBLE,
                     "ColorCtl (round)",
                   0, 40, 20, CCS_ROUND | CCS_RECT | WS_CHILD | WS_VISIBLE,
                     "ColorCtl (round rect)"
                  };

static struct                                 /* dialog control IDs */
{
  WORD  wIDScroll;
  WORD  wIDEdit;
}
    CtlRGB[] = { { IDRED,   IDREDIT },
                 { IDGREEN, IDGEDIT },
                 { IDBLUE,  IDBEDIT } };

static struct                                 /* user-defined styles */
{
  WORD  wFlag;
  char  szName[16];
}
    UserStyle[] = { { CCS_RECT,  "CCS_RECT" },
                    { CCS_ROUND, "CCS_ROUND" } };

/*** FUNCTION PROTOTYPES ***/

GLOBALHANDLE PASCAL FAR ColorCtlInfo( void );
GLOBALHANDLE PASCAL FAR ColorCtlStyle( HWND, GLOBALHANDLE,
                                        LPFNSTRTOID, LPFNIDTOSTR );
int PASCAL FAR          ColorCtlFlags( WORD, LPSTR, WORD );
BOOL PASCAL FAR         ColorCtlDlgFn( HWND, WORD, WORD, LONG );

static void             MsgCommand( HWND, GLOBALHANDLE, WORD, LONG );
static void             MsgClicked( HWND, WORD );
static void             MsgHScroll( HWND, WORD, LONG );
static void             RedisplaySampleColor( HWND );
static void             SetStyleInfo( HWND, LPCTLSTYLE );
static void             GetStyleInfo( HWND, LPCTLSTYLE );
```

(continued)

Figure 4-5. *continued*

```
/***********************************************************************
 *                                                                     *
 * ColorCtlInfo                                                        *
 *                                                                     *
 ***********************************************************************/

GLOBALHANDLE PASCAL FAR ColorCtlInfo()
{
    GLOBALHANDLE    hCtlInfo;
    LPCTLINFO       lpCtlInfo;

    hCtlInfo = GlobalAlloc( GHND, (DWORD)sizeof(CTLINFO) );

    if( hCtlInfo )
    {
      lpCtlInfo = (LPCTLINFO)GlobalLock( hCtlInfo );
      _fmemcpy( lpCtlInfo, &CtlInfo, sizeof(CTLINFO) );
      GlobalUnlock( hCtlInfo );
    }

    return hCtlInfo;
}

/***********************************************************************
 *                                                                     *
 * ColorCtlStyle                                                       *
 *                                                                     *
 ***********************************************************************/

GLOBALHANDLE PASCAL FAR
ColorCtlStyle( HWND hWnd, GLOBALHANDLE hCtlStyle,
                 LPFNSTRTOID lpfnStrToId, LPFNIDTOSTR lpfnIdToStr )
{
    int     nRVal;

    /* save callback function addresses */
    pStrToIdFn = lpfnStrToId;
    pIdToStrFn = lpfnIdToStr;

    /* display the style dialog box */
    nRVal = DialogBoxParam( hDLLInst,
                            "ColorCtlStyleDlg",
                            hWnd,
                            ColorCtlDlgFn,
                            MAKELONG(hCtlStyle, 0) );

    return nRVal;
}
```

(continued)

Figure 4-5. *continued*

```
/**********************************************************************
 *                                                                    *
 * ColorCtlFlags                                                      *
 *                                                                    *
 **********************************************************************/

int PASCAL FAR ColorCtlFlags( WORD wFlags, LPSTR pString, WORD wMaxLen )
{
    int     n;

    /* start with a null string */
    pString[0] = 0;

    /* copy flag string for each user style */
    for( n=0; n<ARRN(UserStyle); n++ )
      if( wFlags & UserStyle[n].wFlag )
      {
        if( *pString )
          lstrcat( pString, " | " );          /* append a separator */

        lstrcat( pString, UserStyle[n].szName );
      }

    /* return the total string length */
    return lstrlen( pString );
}

/**********************************************************************
 *                                                                    *
 * ColorCtlDlgFn                                                      *
 *                                                                    *
 **********************************************************************/

BOOL PASCAL FAR
ColorCtlDlgFn( HWND hDlg, WORD wMsg, WORD wParam, LONG lParam )
{
    static GLOBALHANDLE hCtlStyle;
    LPCTLSTYLE          lpCtlStyle;
    BOOL                bRVal = TRUE;

    switch( wMsg )
    {
      case WM_INITDIALOG:
        hCtlStyle = LOWORD(lParam);
        lpCtlStyle = (LPCTLSTYLE)GlobalLock( hCtlStyle );
        SetStyleInfo( hDlg, lpCtlStyle );
        GlobalUnlock( hCtlStyle );
        break;
```

(continued)

Figure 4-5. *continued*

```
        case WM_COMMAND:
          MsgCommand( hDlg, hCtlStyle, wParam, lParam );
          break;

        case WM_HSCROLL:
          MsgHScroll( hDlg, wParam, lParam );
          break;

        default:
          bRVal = FALSE;
          break;
    }

    return bRVal;
}

/************************************************************************
*                                                                       *
* MsgCommand                                                            *
*                                                                       *
* Notes:                                                                *
*   IDREDIT = (IDRED+1000)                                              *
*   IDGEDIT = (IDGREEN+1000)                                            *
*   IDBEDIT = (IDBLUE+1000)                                             *
*                                                                       *
************************************************************************/

static void
MsgCommand( HWND hDlg, GLOBALHANDLE hCtlStyle, WORD wParam, LONG lParam )
{
    LPCTLSTYLE  lpCtlStyle;
    DWORD       dwRVal;
    char        szID[20];
    HWND        hCtl;
    int         n;

    switch( wParam )
    {
      case IDOK:

        /* verify the control ID */
        GetDlgItemText( hDlg, IDID, szID, sizeof szID );
        dwRVal = (*pStrToIdFn)( szID );

        /* if the control ID is valid, end the dialog */
        if( LOWORD(dwRVal) )
        {
          lpCtlStyle = (LPCTLSTYLE)GlobalLock( hCtlStyle );
```

(continued)

Figure 4-5. *continued*

```
            lpCtlStyle->wId = HIWORD(dwRVal);
            GetStyleInfo( hDlg, lpCtlStyle );

            GlobalUnlock( hCtlStyle );

            EndDialog( hDlg, TRUE );
          }

          else
            PostMessage( hDlg,
                         WM_NEXTDLGCTL,
                         GetDlgItem( hDlg, IDID ),
                         (LONG)TRUE );
          break;

        case IDCANCEL:
          EndDialog( hDlg, FALSE );
          break;

        case IDREDIT:
        case IDGEDIT:
        case IDBEDIT:
          if( EN_KILLFOCUS == HIWORD(lParam) )
          {
            /* verify the new color value */
            n = GetDlgItemInt( hDlg, wParam, NULL, FALSE );
            if( n > MAXCVAL )
            {
              n = MAXCVAL;
              SetDlgItemInt( hDlg, wParam, MAXCVAL, FALSE );
            }

            /* update the corresponding scroll bar */
            hCtl = GetDlgItem( hDlg, wParam-1000 );
            SetScrollPos( hCtl, SB_CTL, n, TRUE );

            RedisplaySampleColor( hDlg );
          }
          break;

        case IDBORDER:
        case IDRECT:
        case IDROUND:
          if( BN_CLICKED == HIWORD(lParam) )
            MsgClicked( hDlg, wParam );
          break;
```

(continued)

Figure 4-5. *continued*

```
      default:
        break;
    }
}

/***********************************************************************
*                                                                      *
* MsgClicked                                                           *
*                                                                      *
***********************************************************************/

static void MsgClicked( HWND hDlg, WORD wParam )
{
    HWND    hCtl;
    DWORD   dwStyle, dwFlag;

    /* determine which style flag to update */
    switch( wParam )
    {
      case IDBORDER:
        dwFlag = WS_BORDER;
        break;

      case IDRECT:
        dwFlag = CCS_RECT;
        break;

      case IDROUND:
        dwFlag = CCS_ROUND;
        break;
    }

    /* update the control style */
    hCtl = GetDlgItem( hDlg, IDCC );
    dwStyle = GetWindowLong( hCtl, GWL_STYLE );

    if( IsDlgButtonChecked( hDlg, wParam ) )
      dwStyle |= dwFlag;
    else
      dwStyle &= ~dwFlag;
    SetWindowLong( hCtl, GWL_STYLE, dwStyle );

    /* redisplay the sample */
    InvalidateRect( hCtl, NULL, TRUE );
    UpdateWindow( hCtl );
}
```

(continued)

Figure 4-5. *continued*

```
/*********************************************************************
*                                                                    *
* MsgHScroll                                                         *
*                                                                    *
*********************************************************************/

static void MsgHScroll( HWND hDlg, WORD wParam, LONG lParam )
{
    int     nPos;
    BOOL    bUpdate = TRUE;

    /* compute a new scroll-bar thumb position */
    nPos = GetScrollPos( HIWORD(lParam), SB_CTL );

    switch( wParam )
    {
      case SB_TOP:
        nPos = 0;
        break;

      case SB_BOTTOM:
        nPos = MAXCVAL;
        break;

      case SB_LINEUP:
        if( nPos > 0 )
          --nPos;
        break;

      case SB_LINEDOWN:
        if( nPos < MAXCVAL )
          ++nPos;
        break;

      case SB_PAGEUP:
        nPos = max( 0, nPos-0x10 );
        break;

      case SB_PAGEDOWN:
        nPos = min( MAXCVAL, nPos+0x10 );
        break;
```

(continued)

Figure 4-5. *continued*

```
      case SB_THUMBPOSITION:
      case SB_THUMBTRACK:
        nPos = LOWORD(lParam);
        break;

      default:
        bUpdate = FALSE;
        break;
    }

    /* update the ScrollBar, Edit, and ColorCtl controls */
    if( bUpdate )
    {
      SetScrollPos( HIWORD(lParam), SB_CTL, nPos, TRUE );
      SetDlgItemInt( hDlg, 1000+GetDlgCtrlID( HIWORD(lParam) ),
                     nPos, FALSE );
      RedisplaySampleColor( hDlg );
    }
}

/**************************************************************************
*                                                                         *
* RedisplaySampleColor                                                    *
*                                                                         *
**************************************************************************/

static void RedisplaySampleColor( HWND hDlg )
{
    BYTE    cValue[3];
    char    szRGB[12];
    HWND    hCtl;
    int     n;

    for( n=0; n<3; n++ )
      cValue[n] = (BYTE)GetDlgItemInt( hDlg, CtlRGB[n].wIDEdit,
                                       NULL, FALSE );
    wsprintf( szRGB, "0x%06lX", RGB(cValue[0], cValue[1], cValue[2]) );

    hCtl = GetDlgItem( hDlg, IDCC );
    SetWindowText( hCtl, szRGB );
    InvalidateRect( hCtl, NULL, FALSE );
    UpdateWindow( hCtl );
}
```

(continued)

Figure 4-5. *continued*

```
/************************************************************************
*                                                                       *
* SetStyleInfo                                                          *
*                                                                       *
************************************************************************/

static void SetStyleInfo( HWND hDlg, LPCTLSTYLE lpCtlStyle )
{
    char    szID[20];
    DWORD   dwRGB;
    BYTE    cValue[3];
    HWND    hCtl;
    int     n;

    /* display the ID value */
    (*pIdToStrFn)( lpCtlStyle->wId, szID, sizeof szID );
    SetDlgItemText( hDlg, IDID, szID );

    /* get the RGB color value from the window text */
    if( lstrcmpi( lpCtlStyle->szTitle, "Text" ) )
      dwRGB = HexToDWord( lpCtlStyle->szTitle );
    else
      dwRGB = DEFRGB;

    cValue[0] = GetRValue(dwRGB);
    cValue[1] = GetGValue(dwRGB);
    cValue[2] = GetBValue(dwRGB);

    /* set the scroll bars and edit controls */
    for( n=0; n<3; n++ )
    {
      hCtl = GetDlgItem( hDlg, CtlRGB[n].wIDScroll );
      SetScrollRange( hCtl, SB_CTL, 0, MAXCVAL, TRUE );
      SetScrollPos( hCtl, SB_CTL, cValue[n], TRUE );

      SetDlgItemInt( hDlg, CtlRGB[n].wIDEdit, cValue[n], FALSE );
    }

    /* display the style check boxes */
    CheckDlgButton( hDlg, IDBORDER,
                  (WS_BORDER == (lpCtlStyle->dwStyle & WS_BORDER)) );
    CheckDlgButton( hDlg, IDRECT,
                  (CCS_RECT == ((WORD)lpCtlStyle->dwStyle & CCS_RECT)) );
    CheckDlgButton( hDlg, IDROUND,
                  (CCS_ROUND == ((WORD)lpCtlStyle->dwStyle & CCS_ROUND)) );
```

(continued)

Figure 4-5. *continued*

```
    /* display the sample control */
    hCtl = GetDlgItem( hDlg, IDCC );

    SetWindowLong( hCtl, GWL_STYLE, lpCtlStyle->dwStyle );
    SetWindowText( hCtl, lpCtlStyle->szTitle );
}

/**************************************************************************
 *                                                                        *
 * GetStyleInfo                                                           *
 *                                                                        *
 **************************************************************************/
static void GetStyleInfo( HWND hDlg, LPCTLSTYLE lpCtlStyle )
{
    BYTE     cValue[3];
    DWORD    dwRGB;
    int      n;

    /* get the color values from the edit controls */
    for( n=0; n<3; n++ )
      cValue[n] =
              (BYTE)GetDlgItemInt( hDlg, CtlRGB[n].wIDEdit, NULL, FALSE );
    dwRGB = RGB(cValue[0], cValue[1], cValue[2] );

    /* use the RGB value as the window text */
    wsprintf( lpCtlStyle->szTitle, "0x%06lX", dwRGB );

    /* update the style flags */
    lpCtlStyle->dwStyle =
            GetWindowLong( GetDlgItem( hDlg, IDCC ), GWL_STYLE );
}
```

```
/**************************************************************************
 *                                                                        *
 * WEP.C                                                                  *
 *   Windows exit procedure for COLORCTL.DLL                              *
 *                                                                        *
 * Exports:        WEP RESIDENTNAME                                       *
 *                                                                        *
 **************************************************************************/

#define NOCOMM
#include    <windows.h>
```

(continued)

Figure 4-5. *continued*

```
/*******************************************************************
*                                                                  *
* WEP                                                              *
*                                                                  *
*******************************************************************/

int PASCAL FAR WEP( int nParam )
{
    return 1;
}

/******************************************************************
*                                                                 *
* COLORCTL.RC resource script                                     *
*                                                                 *
******************************************************************/


#include    <windows.h>
#include    "colorctl.h"


ColorCtlStyleDlg DIALOG LOADONCALL MOVEABLE DISCARDABLE 18, 30, 242, 102
CAPTION "ColorCtl Control Style ..."
STYLE WS_CAPTION | WS_POPUP
{
  CONTROL "&ID:", 0, "Static", SS_RIGHT | WS_CHILD, 4, 6, 12, 8
  CONTROL "", IDID, "Edit", ES_LEFT | ES_UPPERCASE | WS_BORDER
    | WS_TABSTOP | WS_CHILD, 20, 4, 76, 12
  CONTROL "", IDCC, "ColorCtl", CCS_RECT | WS_CHILD, 194, 4, 40, 18
  CONTROL "Color", 0, "Button", BS_GROUPBOX | WS_CHILD, 2, 16, 182, 64
  CONTROL "&Red", 0, "Static", SS_LEFT | WS_CHILD, 6, 30, 24, 8
  CONTROL "", IDRED, "ScrollBar", SBS_HORZ | WS_TABSTOP | WS_CHILD,
    32, 29, 128, 10
  CONTROL "", IDREDIT, "Edit", ES_LEFT | WS_BORDER | WS_TABSTOP | WS_CHILD,
    162, 28, 18, 12
  CONTROL "&Green", 0, "Static", SS_LEFT | WS_CHILD, 6, 46, 24, 8
  CONTROL "", IDGREEN, "ScrollBar", SBS_HORZ | WS_TABSTOP | WS_CHILD,
    32, 45, 128, 10
  CONTROL "", IDGEDIT, "Edit", ES_LEFT | WS_BORDER | WS_TABSTOP | WS_CHILD,
    162, 44, 18, 12
  CONTROL "&Blue", 0, "Static", SS_LEFT | WS_CHILD, 6, 62, 24, 8
  CONTROL "", IDBLUE, "ScrollBar", SBS_HORZ | WS_TABSTOP | WS_CHILD,
    32, 61, 128, 10
  CONTROL "", IDBEDIT, "Edit", ES_LEFT | WS_BORDER | WS_TABSTOP | WS_CHILD,
    162, 60, 18, 12
```

(continued)

Figure 4-5. *continued*

```
   CONTROL "Style", 0, "Button", BS_GROUPBOX ¦ WS_CHILD, 187, 27, 52, 54
   CONTROL "B&order", IDBORDER, "Button", BS_AUTOCHECKBOX ¦ WS_TABSTOP
     ¦ WS_CHILD, 190, 36, 48, 12
   CONTROL "R&ect", IDRECT, "Button", BS_AUTOCHECKBOX ¦ WS_TABSTOP
     ¦ WS_CHILD, 190, 48, 48, 12
   CONTROL "Rou&nd", IDROUND, "Button", BS_AUTOCHECKBOX ¦ WS_TABSTOP
     ¦ WS_CHILD, 190, 60, 48, 12
   CONTROL "Ok", IDOK, "Button", BS_DEFPUSHBUTTON ¦ WS_TABSTOP
     ¦ WS_CHILD, 86, 84, 32, 14
   CONTROL "&Cancel", IDCANCEL, "Button", BS_PUSHBUTTON ¦ WS_TABSTOP
     ¦ WS_CHILD, 142, 84, 32, 14
}

/*************************************************************************
 *                                                                       *
 * COLORCTL.H                                                            *
 *   Header file for COLORCTL.DLL                                        *
 *                                                                       *
 *************************************************************************/

/* ColorCtl styles */
#define CCS_RECT        0x0001L
#define CCS_ROUND       0x0002L

/* dialog control IDs */
#define IDID            100
#define IDCC            101
#define IDRED           102
#define IDREDIT         (1000+IDRED)
#define IDGREEN         103
#define IDGEDIT         (1000+IDGREEN)
#define IDBLUE          104
#define IDBEDIT         (1000+IDBLUE)
#define IDSAMPLE        105
#define IDBORDER        106
#define IDRECT          107
#define IDROUND         108

/* color values */
#define MAXCVAL         0xFF
#define GRAYVAL         0x80
#define DEFRGB          RGB(GRAYVAL,GRAYVAL,GRAYVAL)
```

(continued)

Figure 4-5. *continued*

```
/* miscellany */
#define COLORCTLCLASSNAME   "ColorCtl"
#define ARRN(a)         (sizeof a / sizeof a[0])


/*** FUNCTION PROTOTYPES ***/

/* defined in COLORCTL.C */
LONG PASCAL FAR     ColorCtlWndFn( HWND, WORD, WORD, LONG );
DWORD PASCAL FAR    HexToDWord( LPSTR );

;***********************************************************************
;                                                                      *
; COLORCTL.DEF module-definition file                                  *
;                                                                      *
;***********************************************************************

LIBRARY         COLORCTL
DESCRIPTION     'COLORCTL version 1.0'
STUB            'WINSTUB.EXE'
EXETYPE         WINDOWS

CODE            LOADONCALL MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE SINGLE

HEAPSIZE        1024

SEGMENTS        INIT_TEXT       PRELOAD DISCARDABLE
                WEP_TEXT        PRELOAD FIXED

EXPORTS         WEP             @1 RESIDENTNAME
                ColorCtlInfo    @2
                ColorCtlStyle   @3
                ColorCtlFlags   @4
                ColorCtlWndFn   @5
                ColorCtlDlgFn   @6
```

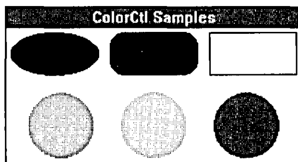


Figure 4-6.
Six ColorCtl *controls with different shapes and colors.*

The basic structural components of COLORCTL.DLL—*LibMain* (which initializes the library and registers the class), the class window function *ColorCtlWndFn*, and *WEP*—represent only about a third of the DLL's source code. This source code is found in INIT.C, COLORCTL.C, and WEP.C. The rest of the source code, in DLGEDIT.C, provides the resource-editor interface.

Initialization and Class Registration

The library's initialization function, *LibMain*, registers the custom-control class with the CS_GLOBALCLASS style. The control class also has the CS_DBLCLKS style so that you will be able to double-click the control within a resource editor to view the control's style dialog box. The initialization function also stores the library's module handle in a global variable. The module handle is later used in DLGEDIT.C to identify a dialog resource compiled as part of the library.

The Class Window Function

The window function for the *ColorCtl* class is named *ColorCtlWndFn* and is defined in COLORCTL.C. The window function contains no additional code in support of the resource-editor interface—there is no difference between a *ColorCtl* control created by a resource editor and one created by any other application.

The DLL Exit Function

As it is in all DLLs, the exit function is named *WEP* and is exported with the RESIDENTNAME attribute. Unlike the RYG.DLL example earlier in this chapter, COLORCTL.DLL has no cleanup actions to carry out when Windows unloads the library. Therefore *WEP* is a do-nothing function.

Three of the other four exported functions in COLORCTL.DLL provide the resource-editor interface. These functions are *ColorCtlInfo*, *ColorCtlStyle*, and *ColorCtlFlags*. The fourth, *ColorCtlDlgFn*, supports the dialog box displayed when the resource editor calls *ColorCtlStyle*.

The *Info* Function

A resource editor calls the *Info* function to determine the characteristics of the custom-control class. For example, when you choose the Add Custom Control command from the SDK Dialog Editor's File menu and then specify *ColorCtl* in

the Add Control dialog box, the Dialog Editor calls *ColorCtlInfo* to determine the default class name, size, and style of the control.

The *Info* function returns a handle to a global-memory block that contains descriptive information for the custom-control class. This information is formatted as a CTLINFO data structure. (This data structure is declared in CUSTCNTL.H, a C-language include file provided in the Windows SDK.) The SDK Dialog Editor refers to the CTLINFO data whenever the editor creates a new custom-control window.

The *Style* Function

A resource editor calls the *Style* function to let you specify the style of a particular custom control. In the SDK Dialog Editor, this happens when you choose the Style command from the Edit menu or when you double-click on a control that has the CS_DBLCLKS style. The Style command displays a dialog box that lets you edit a particular control's *CreateWindow* parameters. A resource editor calls the *Style* function with a CTLSTYLE data structure containing a set of default style parameters. The *Style* function modifies these parameters according to your input in the dialog box.

In the *ColorCtl* example, the style function *ColorCtlStyle* displays a modal dialog box that lets you specify the control's color and border styles. The return value from the call to *DialogBoxParam* indicates whether you have modified the values in the CTLSTYLE data structure.

Although most of a control's style attributes can be modified at your discretion, the control's ID requires special handling to ensure that its value is unique among the controls that are used within a dialog box. Typically, a resource editor assigns a unique default ID value to each new control. The resource editor then lets the *Style* function validate the ID value by passing the addresses of two callback functions as parameters. The *Style* function can call the first of the callback functions (*lpfnIdToStr*) to discover the ID value that the resource editor has assigned. If you change the control's ID value, the *Style* function can call the other callback function (*lpfnStrToId*) to inform the resource editor of the new ID value and to allow the editor to verify that the changed ID value is unique.

The *Flags* Function

The *Flags* function builds a string of symbols that represent the style of a custom control. A resource editor uses the string returned by the *Flags* function to build a CONTROL statement in a resource-script file.

The *ColorCtlFlags* function uses the *wFlags* parameter to build a text string containing the appropriate style names separated by vertical bars (the logical OR operator used in a CONTROL statement). The function returns the final length of the text string. Although *ColorCtlFlags* builds the string without monitoring its length, a *Flags* function that builds a long text string should verify that the string length does not exceed the maximum buffer size specified in the *wMaxLen* parameter.

The Style Dialog Function

The style dialog function supports a dialog box displayed by the *Style* function. The dialog's purpose is to let you update the values in the CTLSTYLE data structure for a particular custom-control window. In general, this dialog box should contain both OK and Cancel buttons. If you choose OK, the *Style* function should return updated CTLSTYLE data to the resource editor. If you choose Cancel, the CTLSTYLE values should remain unmodified.

The *ColorCtlDlgFn* function supports the dialog box illustrated in Figure 4-7. If you choose the OK button, the function first updates the CTLSTYLE values by calling *GetStyleInfo* and then calls *EndDialog* with a nonzero return value. It calls *EndDialog* with a return value of 0 if you choose Cancel or if you use the Esc key to end the dialog.

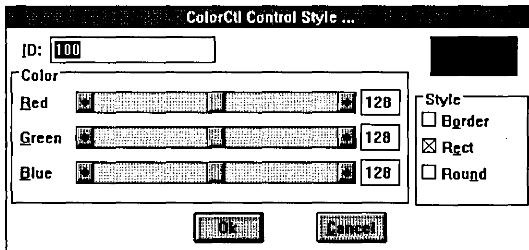


Figure 4-7.
The dialog box that is displayed when the Dialog Editor calls the ColorCtlStyleFn *function.*

Building the Perfect Control

When you intend to use a custom control as a general-purpose tool, you need to give careful thought to the control's visual design and to the details of its functionality. Like the predefined Windows controls, custom controls should blend into the Windows environment so that they cooperate well with their parent

windows as well as with other controls. As you fine-tune a custom-control class, you may want to adopt some of the techniques that Windows' predefined control classes use for memory management, client-area painting, and communicating with a parent window.

Managing the Input Focus

A control should provide a visual indication that it has the input focus. For example, edit controls highlight their contents; list-box controls outline the currently selected item with a dotted-line rectangle; scroll-bar controls display a blinking caret in the scroll-bar thumb; and button controls display a modified border and a dotted-line rectangle around the button text. If your custom control can use the same focus indication as one of the default control classes, users will intuitively recognize when your custom control has the input focus.

Memory Allocation

There are several different ways for a custom control to allocate memory. Because controls are windows, you can store limited amounts of data in window extra bytes and window property lists. If a control needs more than a few bytes of data, however, it should call *LocalAlloc* or *GlobalAlloc* to obtain a block of memory. It can then store the block's memory handle in window extra bytes.

If you use *LocalAlloc*, be sure you know which local heap you are using. Normally, calling *LocalAlloc* from a library function allocates memory in the library's local heap. If you use a DLL's local heap, be certain to specify a nonzero value in a HEAPSIZE statement in the DLL's module-definition file.

If a custom control must process long lists of data elements, you might also want to allow the control's owner to allocate memory and to pass memory handles to the control. This is a technique that is used by edit controls, which use the class-specific EM_SETHANDLE and EM_GETHANDLE messages to manage memory handles.

Notifying the Parent Window

You should also consider using one or more messages to let a custom control notify its parent when the control's status changes. The method used by the predefined Windows controls is to send WM_COMMAND to the parent, with the control ID in *wParam*, the control's window handle in the low-order word of *lParam*, and a notification code in the high-order word of *lParam*.

A custom control that accepts keyboard input might send a WM_COMMAND message to its parent whenever it processes a keystroke. The EN_CHANGE notification code sent by an edit control is an example of this. A custom control could also support an owner-draw style by sending WM_DRAWITEM messages to its parent, as do button, list-box, and combo-box controls that have owner-draw styles.

A control can provide an additional degree of flexibility by sending WM_CTL-COLOR messages to its parent. The method for sending WM_CTLCOLOR is shown in the function *MsgEraseBkgnd* in the file COLORCTL.C. The control's window function sends WM_CTLCOLOR to the parent in response to WM_ERASEBKGND, which is sent to the control's window function by *Begin-Paint.* The device-context handle associated with WM_ERASEBKGND is the same as the one returned by *BeginPaint,* and the control's window function uses this handle when it sends WM_CTLCOLOR.

The WM_CTLCOLOR message is useful for *ColorCtl* controls because it lets a control's parent determine how to paint the control's background. If the parent lets *DefWindowProc* process WM_CTLCOLOR, the control's background color will be the same as the default window background. However, if the parent processes WM_CTLCOLOR by returning a handle to a brush, the control will use that brush to paint its background. (This works even if the parent window is a dialog box.) The parent of a *ColorCtl* control can also prevent the control from erasing its background by returning the handle of a null brush in response to WM_CTLCOLOR.

Painting Control Styles

Whenever you design a custom control, pay particular attention to the way the control is painted, especially if the control supports more than one visual variant. Sometimes a control's final design reflects a compromise between visual consistency and functionality. For example, if you design a control that displays text, you may want the text to be displayed at the same location in each control window regardless of whether the control displays a border. On the other hand, it may be more convenient to work with a text-display control in which the position of the text can vary to minimize the amount of space necessary to display the text. Figure 4-8 illustrates the latter design in two edit controls, only one of which has the WS_BORDER style.

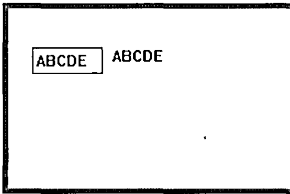


Figure 4-8.
The edit control on the left has the WS_BORDER style; the one on the right does not. Both controls are the same size, but the window text is aligned differently in each.

In the case of the *ColorCtl* custom control, the visual design derives from the control's purpose, which is to fill itself with a color. The control displays a rectangle, an ellipse, or a round-cornered rectangle whose size is bounded by the control's window rectangle, not by the client area. This design prevents gaps between adjoining *ColorCtl* controls and allows you to align or overlap multiple *ColorCtl* controls without needing to adjust the controls' size or border style.

The control's visual design depends on selective processing of the WM_NCPAINT and WM_PAINT messages. For a window with the WS_BORDER style, Windows' default action is to draw a rectangular black border in response to the WM_NCPAINT message. Because this obviously will not work for *ColorCtl* controls that have non-rectangular styles, the *ColorCtl* window function traps WM_NCPAINT and paints the border itself in response to WM_PAINT messages. This is convenient because the window function calls *Rectangle*, *Ellipse*, or *RoundRect* to paint the control's client area, and these GDI functions can draw a border as well as fill the client area with color.

For this approach to work, however, the control window's client area must always correspond to the entire control-window rectangle. This is a problem if the window has the WS_BORDER style because Windows' default action is to shrink the window's client area to leave room for the border. For this reason, the window function traps WM_NCCALCSIZE, which prevents *DefWindowProc* from reducing the client-area size for controls with the WS_BORDER style.

Although this technique seems straightforward, it creates an additional complication for *ColorCtl* controls that do not have the WS_BORDER style. The problem is that the *Rectangle*, *Ellipse*, and *RoundRect* functions always draw a border, even if you select a null pen for the current device context. The solution is to increase the size of the drawing rectangle with a call to *InflateRect*. This

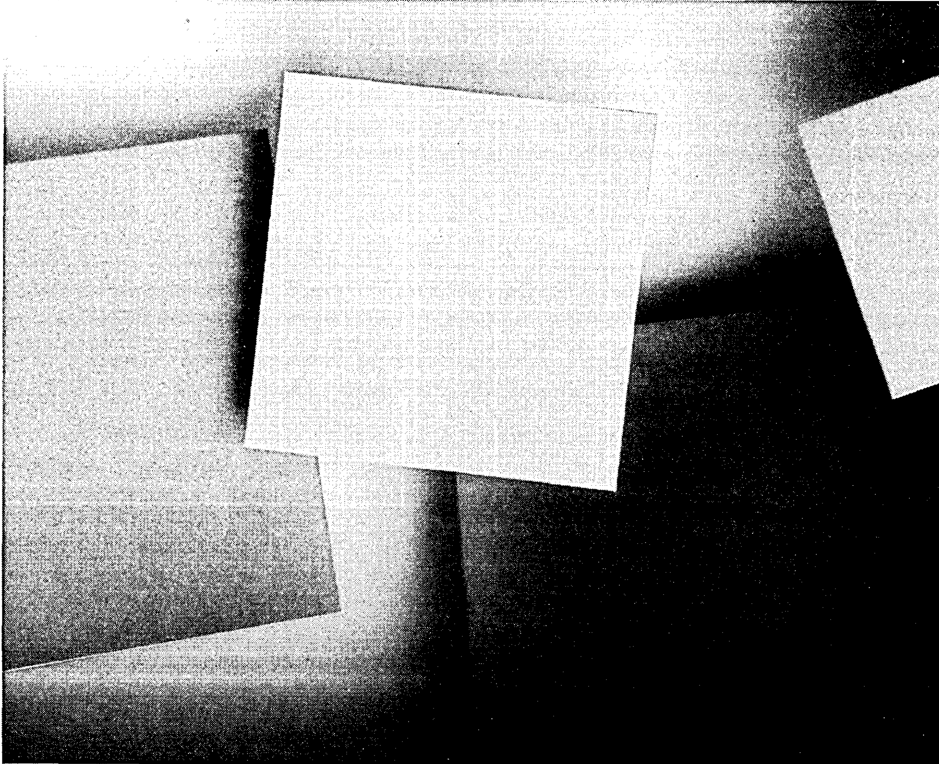
causes the border to be clipped outside the control's client area, so the filled interior of the rectangle, ellipse, or round-cornered rectangle extends to the edge of the control's window rectangle.

These complications could be avoided by restricting the *ColorCtl* control's style to rectangles with a border. However, the control class is more powerful because it supports variations in its border and shape styles. The price of additional functionality in a custom-control class is usually additional programming effort.

You may find that the goal of building a perfect custom control is an elusive one. If you reflect on the *RYG* and *ColorCtl* examples earlier in this chapter, you will probably think of ways to improve their appearance or their functionality. For example, both control classes would benefit from the ability to respond to user input. In fact, there's no end to the tinkering you might do to make these custom controls—or any other custom control—a bit more flexible or prettier to look at. Nonetheless, the extra effort is worthwhile. Well-designed custom controls make Windows applications easier to write and easier to use.

5

An Object-Oriented View



If you program in an object-oriented environment such as Smalltalk-80, you'll be happy to learn that the Windows environment contains some familiar object-oriented design ideas, but you'll be disappointed that Windows doesn't support object-oriented programming constructs more fully. Even if you're more comfortable with procedural programming and unfamiliar with object-oriented concepts, understanding Windows' object-oriented roots can help you design better source code for your Windows applications.

This chapter looks at Windows from an object-oriented point of view. Windows itself is not an object-oriented programming environment, but the underlying structure of the Windows environment is clearly influenced by object-oriented software concepts. This chapter describes those concepts and shows how you can use them in the design of your own Windows applications.

Objects and Messages

The term "object" has different meanings in different contexts. By now, you're surely familiar with global-memory objects, local-memory objects, and GDI objects. You have also encountered object libraries and object modules, in which an object is something generated by a compiler or an assembler. In this chapter, however, an object is none of these. Instead, an object is a particular kind of programming construct used in a style of programming called object-oriented programming.

Object Structure

A typical object-oriented programming environment consists of a variety of predefined objects arranged so that they can transfer control to each other in a hierarchical fashion. Each of these objects consists of both executable code and data. The executable code describes a set of predefined actions the object can perform. The data is private to the object—that is, it is accessed only by the associated executable code. In the parlance of object-oriented programming, this localization of predefined actions and private data within an object is called encapsulation.

From this point of view, an object is not a complicated entity. You can easily imagine a straightforward C function that uses a *switch* statement to define a set of actions that access data stored in locations known only within the function itself. (If you think this description sounds suspiciously like a Windows window, you're absolutely correct.)

A natural characteristic of the design of an object-oriented environment is that objects transfer control to each other using messages. A message is represented as a set of data items that can be transferred between objects. Sending a message is equivalent to executing a function call with parameters that represent the message data. One of the parameters in this function call is a predefined data item that identifies the message. When an object receives a message—that is, when the executable function that performs an object's actions is called with a message identifier and other parameters—the message identifier determines which action the object carries out.

Using Messages

As a Windows programmer, you are already familiar with the technique of using messages to evoke actions. The power of this technique lies in the fact that different objects can respond to the same message with different actions. This means that a particular message can represent a single generic event such as a keystroke, mouse movement, or video-display update, yet any particular message can evoke quite different actions in different objects. There are many obvious examples of this in the Windows environment: Consider the different ways in which windows can process the same WM_KEYDOWN, WM_MOUSEMOVE, or WM_PAINT message.

It's harder to follow a program's flow in a message-driven environment than in a procedural operating environment. There are at least two reasons why this is so. One is that messages can be sent to an object either by another object or by the operating environment itself. In Windows, for example, messages such as WM_KEYDOWN and WM_MOUSEMOVE originate within the operating environment, whereas messages such as WM_SETFONT originate within applications.

Another reason is that an object can process a message by sending one or more new messages to other objects or even to itself. If you are analyzing program flow, you may find that carrying out even a simple action involves processing a cascade of messages. This is why debugging a Windows application often involves monitoring the messages the application processes.

Although the order in which an object receives messages is somewhat unpredictable, the actions taken by the object to process each message are explicitly defined. Of course, objects do not explicitly define actions for every possible message. Instead, objects rely on an implicit default action to be carried out for any message not explicitly processed. The usual mechanism that objects use for default message processing is to pass messages to another object.

Message-Passing Hierarchy

In an object-oriented programming system, objects are related to each other in a hierarchical structure that reflects the flow of messages between the objects. When an object receives a message, one of its possible actions may be to retransmit the message to another object that lies above it in the hierarchy. A message-passing hierarchy implies that an object can always respond to a message with a default action—namely, the action performed by the next object in the hierarchy.

The topmost objects in an object hierarchy are naturally the most generic in their actions because they provide the default actions for any messages passed to them from objects below them in the hierarchy. In a very simplistic object hierarchy, the topmost object is simply a "black hole" that ignores all messages it receives. In a real-world programming environment, however, the topmost object may carry out a variety of generic default actions.

A message-passing hierarchy is a powerful programming paradigm because an object lower in the hierarchy can make use of the functionality of objects above it in the hierarchy simply by passing messages up the hierarchy. Such use of the hierarchy is characteristic of object-oriented programming systems. It is also implicit in the message-passing mechanisms supported in Windows.

Windows as Objects

In the Windows environment, you can regard windows as objects. Each window is associated with private data and with a window function that explicitly defines a set of actions to be carried out in response to one or more specific messages. If a window function does not explicitly process a message, it passes the message to another window function in a loosely hierarchical fashion.

You can think of *DefWindowProc* as the topmost function in a message-passing hierarchy. *DefWindowProc* carries out a variety of generic actions common to most windows in the Windows environment, such as drawing a window's nonclient area, responding to system commands to resize and move a window, and updating a window's text caption. *DefWindowProc* also serves as a "black hole" for many messages. About one third of the documented WM_* messages are processed by *DefWindowProc* in Windows 3.0. *DefWindowProc* does nothing with the others except return a value of 0.

The messages ignored by *DefWindowProc* are intended for processing by other window functions. Because *DefWindowProc* carries out the basic actions common to most windows, most window functions perform specialized actions in

response to certain messages and pass the remaining messages up the hierarchy to *DefWindowProc* for default processing.

Message-Passing in Windows

In a true object-oriented programming environment such as Smalltalk-80, the mechanisms by which objects pass messages to other objects are explicitly defined. In Windows, however, message-passing is implemented as a function call that can occur anywhere within a window function. The way a message passes from one window function to another depends on how the calling window function executes the function call and on which function is called.

In principle, the path of a message through the Windows hierarchy is simple: A message is passed—from window function to window function—until it is trapped and processed by a window function that recognizes it. The problem is that some messages must always be passed up the object hierarchy, whether or not they are processed in a window function, because *DefWindowProc* and other default functions (such as those associated with the predefined control classes) carry out essential actions in response to some messages.

Unfortunately, you don't always know which messages can be safely trapped in a window function and which must be passed through. To be on the safe side, a window function should trap only those messages whose default processing it needs to override. All other messages should be passed up the hierarchy either before or after they are processed in the window function.

This means a window function can handle a message in one of four ways. The window function can trap the message without passing it up the object hierarchy; it can process the message and then pass it on; it can pass the message up the hierarchy and then process it; or it can simply pass the message on.

Figure 5-1 illustrates all four kinds of message processing. The function *NumEditWndFn* supports an edit-class window that recognizes only numeric input. The function passes most message's up the object hierarchy to the default edit window function by calling the *CallWindowProc* function, but it traps WM_CHAR messages by setting the *bCWP* variable to FALSE for any character that does not represent numeric input. The *NumEditWndFn* function also processes the WM_SETFOCUS message after the message has been transmitted to the default edit window function. In this way, *NumEditWndFn* displays a special caret instead of the caret created by the default edit window function in response to WM_SETFOCUS.

```
FARPROC pDefEditWndFn;     /* pointer to the default
                              edit window function */


LONG PASCAL FAR
NumEditWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bCWP = FALSE;


    switch( wMsg )
    {
      case WM_CHAR:
        bCWP = iscntrl(wParam) || isdigit(wParam) ||
                (NULL != strchr( "+-.", wParam ));
        break;

      default:
        bCWP = TRUE;
        break;
    }

    if( bCWP )
      lRVal = CallWindowProc( pDefEditWndFn,
                                      hWnd, wMsg, wParam, lParam );

    if( WM_SETFOCUS == wMsg )
    {
      /* create a fat caret */
      CreateCaret( hWnd, 0, 4, 16 );
      ShowCaret( hWnd );
    }

    return lRVal;
}
```

Figure 5-1.
Source code for NumEditWndFn, *a window function for an edit control that allows only numeric input.*

Classes

Missing from this view of window-objects arranged in a message-passing hierarchy is an efficient way to create a new object with specified functionality in its proper place in the hierarchy. In Windows, this is accomplished by using another object-oriented programming construct: the class.

Class Structure

In an object-oriented environment, a class describes the characteristics of a set of similar objects. In Windows, a class specifies such characteristics as the class name and the address of the window function to be used by all objects (windows) in the class. When you call *CreateWindow*, you specify the class whose characteristics apply to the window being created. This is much more efficient than explicitly specifying all the characteristics of every window you create, as you might if classes did not exist.

By using classes, Windows can create multiple objects—that is, multiple windows—using only one copy of the executable code that defines a class's functions. Different windows within the same class are distinguished only by private data (parent-window handle, child-window ID, window-function address, window extra bytes, and so on) that are associated with each window. Windows assigns a unique handle to each newly created window and uses the handle to identify the window's private data. Applications access a window's private data through the API functions *SetWindowWord*, *GetWindowWord*, *SetWindowLong*, and *GetWindowLong*.

Window classes are also associated with private data. (However, window classes do not process messages, so don't think of them as objects.) You initialize a class's private data in the WNDCLASS data structure that you use with *RegisterClass*. Windows subsequently refers to this data when you call *CreateWindow* to create a window in the class. You can use several API functions—including *GetClassInfo*, *GetClassLong*, *GetClassWord*, *SetClassLong*, and *SetClassWord*—to access a class's private data.

Subclasses

In Windows, window functions implement the hierarchical flow of messages. Because window classes contain the addresses of window functions, you can use classes to describe the message-passing hierarchy within a Windows application. However, there is no class hierarchy in Windows that completely describes the hierarchical flow of messages. Window functions such as *DefWindowProc* do not correspond to any class.

Windows also lacks an intrinsic mechanism for subclassing—that is, for creating a new class that inherits functionality from a class hierarchy. Each time you register a new class, you must explicitly define the private data and window function of the class. You cannot implicitly describe the default functionality of a new class by relying on inheritance from a previously defined class.

Nevertheless, subclassing is a useful programming technique in Windows. To create a subclass, you use *GetClassInfo* to copy a class's private data into a WNDCLASS data structure. You then modify the private data and provide the address of a new window function that passes unprocessed messages to the original class's window function. When you call *RegisterClass* with a new class name and the modified WNDCLASS data structure, you create a new subclass with a unique set of characteristics. The subclass can inherit some or all of the functionality of the original class, depending on how the window function of the subclass passes messages to the window function of the original class. You can then use the subclass to create new windows.

Figure 5-2 illustrates how you might create the *NumEdit* class, a subclass of the default edit control class. The subclass, whose window function is shown in Figure 5-1, recognizes only numeric input. The *NumEdit* subclass inherits all the functionality of the default edit class because the *NumEdit* class window function (*NumEditWndFn*) passes all unprocessed messages to *DefEditWndFn*, the edit-class window function.

```
char    szNumEditClass[] = "NumEdit";
FARPROC pDefEditWndFn;


BOOL RegisterNumEditClass( HANDLE hInstance )
{
  WNDCLASS  wc;


  /* get default WNDCLASS values for the edit class */
  GetClassInfo( 0, "Edit", &wc );

  /* save the address of the default edit window function */
  pDefEditWndFn = (FARPROC)wc.lpfnWndProc;

  /* register the NumEdit subclass */
  wc.hInstance       = hInstance;
  wc.lpszClassName   = szNumEditClass;
  wc.lpfnWndProc     = NumEditWndFn;

  return RegisterClass( &wc );
}
```

Figure 5-2.
Creating a subclass of the edit control class. The subclass is named NumEdit*; the subclass window function is named* NumEditWndFn.

From this object-oriented perspective, what is sometimes called subclassing in Windows isn't really subclassing. The Windows SDK documentation uses the term "subclassing" to describe a different technique that creates "subclasses" on a window-by-window basis. This technique, shown in Figure 5-3, associates a new window function with a particular window by using *GetWindowLong* to steal the address of the window's original window function from the window's private data and then calling *SetWindowLong* to redirect the window's messages to a new window function that filters some of the messages. In effect, filtering messages in this way changes a window's location in Windows' message-passing hierarchy, but it does not actually involve the creation of a new class or subclass.

```
FARPROC  pDefEditWndFn;


void InstallFilterFunction( HWND hEdit )
{
  FARPROC  pThunk;


  pThunk = MakeProcInstance( (FARPROC)NumEditWndFn, hInstance );
  pDefEditWndFn =
       (FARPROC)SetWindowLong( hEdit, GWL_WNDPROC, (LONG)pThunk );
}


void UninstallFilterFunction( HWND hEdit )
{
  FARPROC  pThunk;


  pThunk = (FARPROC)SetWindowLong( hEdit, GWL_WNDPROC,
       (LONG)pDefEditWndFn );
  FreeProcInstance( pThunk );
}
```

Figure 5-3.
Installing and uninstalling a window function that filters messages without creating a new window subclass. In this example, the function NumEditWndFn *is installed to filter the messages sent to the edit-class window* hEdit.

Classes and the Appearance of Objects

Apart from their convenience as a programming construct, classes intuitively describe the elements of Windows' graphical interface. By predefining a set of useful classes, Windows makes it easy to create windows that have a great deal

of built-in functionality as well as a consistent visual style. It is no coincidence that the predefined control-class names describe the different entities that appear on the screen.

In Windows, characteristics such as an object's visual appearance and default functionality are part of a class description. This creates an intuitive connection between window classes and the visual appearance of windows on the screen. As a programmer, you can think of ListBox as the name that identifies a particular class; as a Windows user, you can easily visualize the corresponding set of list box controls.

Objects and Data

Part of the design of objects in an object-oriented programming environment is the association of private data with each object. You can regard window extra bytes and property lists as two different mechanisms for associating private data with windows as objects.

Window Extra Bytes

When you call *CreateWindow*, Windows allocates a fixed-size data structure that is private to the new window. This data structure contains the window's instance handle, parent-window handle, window-function address, and other data that Windows uses to manage the window. The same data structure can be made larger than the minimum size used by Windows' window manager to store several extra bytes of private data. The extra bytes of data in the data structure are ignored by the window manager and can be used freely by your programs.

Because window extra bytes contain data that is private to a particular window, it makes sense to use them to keep track of data on a window-by-window basis. The only method for manipulating this private data is through the API functions *GetWindowWord*, *GetWindowLong*, *SetWindowWord*, and *SetWindowLong*, which access window extra bytes by using a window handle.

The number of extra bytes allocated for a window is specified by the window's class. This means that, to use window extra bytes, you must first register a window class that specifies the number of window extra bytes to be allocated for windows in the class. When you call *RegisterClass*, the *cbWndExtra* value in the WNDCLASS data structure specifies the number of extra bytes of data to associate with windows in the class. When you subsequently create a window in the class, *CreateWindow* allocates the specified number of extra bytes and associates it with the window's handle.

The nature and format of the data you store in window extra bytes are entirely up to you. However, the API functions are clearly designed to store and retrieve only small chunks of data. If you want to associate more than 6 or 8 bytes of data with a window, you should allocate a block of memory by using *GlobalAlloc* or *LocalAlloc*, store the data in the memory block, and store the memory handle in the window extra bytes:

```
/* allocate a 1-KB block of private data */
hMem = GlobalAlloc( GHND, 1024L );

/* store the handle in the window extra bytes */
SetWindowWord( hWnd, 0, hMem );
```

When you do this, you must manage the handle stored in the window extra bytes. For example, if you call *GlobalReAlloc* to change the size of the memory block, you must also update the value stored in the window extra bytes. Also, you should free the memory block when the window is destroyed:

```
case WM_DESTROY:
  GlobalFree( (GLOBALHANDLE)GetWindowWord( hWnd, 0 ) );
  break;
```

The problem with using window extra bytes is that you must design your application so that the layout of every window's extra bytes is known. This can be inconvenient if you use window extra bytes differently in different window classes. To be smart about the layout of window extra bytes in windows of different classes, your program must determine a window's class—perhaps by a call to *GetClassName*—before it accesses the window's extra bytes.

Using window extra bytes is also problematic if you use subclasses. If you use *GetClassInfo* to create a subclass of a class that uses window extra bytes, the subclass must allocate additional extra bytes so that the original class's extra bytes are not clobbered. Later, when you create a window in the subclass and access its extra bytes, you must skip over the extra bytes used in the original class, as shown in Figure 5-4.

Property Lists

An elegant way to avoid the problems with window extra bytes is to use property lists. Instead of identifying a window's private data items with an offset into a data structure, the property-list API lets you assign names to a window's private data items.

```
int    nEBStart;         /* a global variable */

void RegisterTheSubclass( ... )
{
  WNDCLASS     wc;

  /* save the current number of window extra bytes */
  GetClassInfo( ..., &wc );
  nEBStart = wc.cbWndExtra;

  /* allocate additional extra bytes for the subclass */
  wc.cbWndExtra += sizeof(WORD);
  :
  RegisterClass( &wc );
}

void AccessTheExtraBytes( ... )
{
  /* access the extra bytes (skip the previous allocation) */
  SetWindowWord( hWnd, nEBStart, wData );
  :
  wData = GetWindowWord( hWnd, nEBStart );
}
```

Figure 5-4.
Allocating and using window extra bytes in a subclass. In this example, 2 extra bytes (one word) are allocated for the subclass.

Although the terminology is borrowed from the Lisp language, property lists in Windows are not really the same as property lists in Lisp. In Lisp, a language based on list-processing concepts, properties represent only one of a variety of ways to manipulate lists. In Windows, a property is nothing more than a data item associated with a particular window and identified by name; a property list is a list of a window's properties. You can regard a window's properties as private data items identified by names.

Windows provides a set of straightforward API functions for manipulating properties. To associate a private data item with a window, you use the *SetProp* function:

```
SetProp( hWnd, lpName, hData );
```

The window handle *hWnd* identifies the window; the string *lpName* contains the name of the data item; and *hData* is a handle to a local or global block of memory that contains the data. (Actually, you can use *hData* to represent not only memory handles but any 2-byte data item.) Windows does the internal list processing required to keep track of the names and data handles associated with each window.

To access the data, you call *GetProp* using the window handle and name you passed to *SetProp*:

```
hData = GetProp( hWnd, lpName );
```

To discard the property, you call *RemoveProp*:

```
hData = RemoveProp( hWnd, lpName );
```

Before you destroy a window, you must call *RemoveProp* for each property you have associated with the window.

In general, you should use property lists when you want to name the private data items associated with a window. Property lists are also easier to use than window extra bytes in cases where you want to associate private data with pre-existing windows without usable window extra bytes.

When you use property lists, you must carefully manage both the property data items and the property names. The amount of data directly associated with a property name is only 2 bytes, the size of a Windows handle. This means you must always access a property data item indirectly unless the data item is itself only 1 or 2 bytes long.

One feature of Windows' property-list API is that the same property name can be associated with different windows, regardless of the window class or the application to which a particular window belongs. The catch is that you must be careful to use unique property names when you add to a window's property list. If you call *SetProp* with a pre-existing property name, the function will simply update the data associated with the property name. If you want to ensure that a property name is unique when you use it, call *GetProp* with the new property name before you call *SetProp* to add the property to a window's property list; if *GetProp* returns 0, the property name was not previously associated with the window.

Atoms as Property Names

In some programs, you might find it convenient to use atoms instead of strings as property names. An atom is an unsigned integer value that uniquely identifies a string stored by Windows in a hash table. Windows' atom manager supports both global and local atoms. The hash table for local atoms is stored in a module's local heap; the hash table for global atoms is stored in the global heap.

You can use either a local or a global atom as a property name. To do this, call *AddAtom* or *GlobalAddAtom* to create an atom, and then use the atom instead of a string pointer when you call the property-list API functions. When you call *SetProp*, *GetProp*, and *RemoveProp*, the atom must be passed in the low-order word of the *lpName* parameter, with the high-order word set to 0. You can use the MAKEINTATOM macro to do this conversion:

```
aAtom = GlobalAddAtom( lpName );
SetProp( hWnd, MAKEINTATOM(aAtom), hData );
```

Two Programming Examples

The following section presents two source-code examples that look at Windows from an object-oriented point of view in that they treat windows as objects with private data. These examples use the property-list API to implement functions that might otherwise be considerably more awkward to develop.

Using a Property List

The first example, in Figure 5-5 on the next page, consists of two functions, *ShowWaitCursor* and *HideWaitCursor*. These two functions use the property-list API to associate a cursor handle with a specified window handle. This technique lets a program call the functions with any window handle as the parameter without the need to save and restore the cursor handle in a static variable elsewhere in the program.

ShowWaitCursor uses *LoadCursor* and *SetCursor* to change the current cursor shape to an hourglass. *ShowWaitCursor* calls *SetProp* to add the specified window's previous cursor handle to the window's property list. The string *szPropID* identifies the cursor handle in the property list. The complementary function *HideWaitCursor* calls *RemoveProp* to extract the cursor handle from the property list and to remove the *szPropID* property. *HideWaitCursor* then restores the cursor through a call to *SetCursor*.

```
/* property name */
char szPropID[] = "hPrevCursor";

/**********************************************************************
*                                                                     *
* ShowWaitCursor                                                      *
*                                                                     *
**********************************************************************/

static void ShowWaitCursor( HWND hWnd )
{
    HCURSOR hCursor;


    hCursor = GetProp( hWnd, szPropID );

    if( 0 == hCursor )
    {
      /* display the wait cursor */
      ShowCursor( TRUE );
      hCursor = SetCursor( LoadCursor( 0, IDC_WAIT ) );

      /* save the previous cursor handle in the window's property list */
      SetProp( hWnd, szPropID, hCursor );
    }
}

/**********************************************************************
*                                                                     *
* HideWaitCursor                                                      *
*                                                                     *
**********************************************************************/

static void HideWaitCursor( HWND hWnd )
{
    HCURSOR hCursor;

    /* remove the property from the window's property list */
    hCursor = RemoveProp( hWnd, szPropID );

    if( 0 != hCursor )
    {
      /* display the previous cursor */
      SetCursor( hCursor );
      ShowCursor( FALSE );
    }
}
```

Figure 5-5.
Source code for ShowWaitCursor *and* HideWaitCursor.

Filtering Messages

The second example, shown in Figure 5-6, illustrates an alternative technique for filtering the messages processed by a window function. This technique uses the property-list API to store the address of a window's default window function. The advantage to using a window's property list instead of a static variable to store this address is that you can associate different default window functions with different windows without modifying the window function that does the message filtering.

```
#*************************************************************************
#                                                                        *
# NMAKE description for KEYTRAP.EXE                                      *
#                                                                        *
#*************************************************************************

.c.obj:
    cl /AM /c /G2sw /Osw /W4 /Zlp $*.c

ALL:    keytrap.exe

keytrap.obj:    keytrap.c keytrap.h

keytrap.res:    keytrap.rc keytrap.h keytrap.ico
    rc /r keytrap.rc

keytrap.exe:    keytrap.obj keytrap.res keytrap.def
    link /al:16 /nod /noe keytrap, , , libw mlibcew, keytrap.def
    rc keytrap.res
```

```
/*************************************************************************
*                                                                        *
* KEYTRAP.C                                                              *
*                                                                        *
* Exports:      TopLevelWndFn                                            *
*               KeyTrapWndFn                                             *
*                                                                        *
*************************************************************************/

#define NOCOMM
#include    <windows.h>
#include    "keytrap.h"
```

Figure 5-6.
Source code for KEYTRAP.EXE.

(continued)

Figure 5-6. *continued*

```
/*** FUNCTION PROTOTYPES ***/

typedef struct
{
  FARPROC   pThunk;
  FARPROC   pDefWndFn;
} FNSTRUC;

typedef FNSTRUC NEAR * NPFNSTRUC;

LONG PASCAL FAR TopLevelWndFn( HWND, WORD, WORD, LONG );
LONG PASCAL FAR KeyTrapWndFn( HWND, WORD, WORD, LONG );

static HWND     Init( HANDLE, HANDLE, int );
static void     InstallKeyTrap( HWND );
static void     UninstallKeyTrap( HWND );

/*** GLOBAL VARIABLES ***/

char    szTopLevelClass[] = KEYTRAPCLASSNAME;
char    szAppTitle[] = "Key Trap";
char    szFNStruc[] = "FNStruc";

HANDLE  hInstance;


/*************************************************************************
*                                                                       *
* WinMain                                                               *
*                                                                       *
*************************************************************************/

int PASCAL
WinMain( HANDLE hInst, HANDLE hPrevInst, LPSTR lpszCmdLine, int nCmdShow )
{
    HWND    hWnd;
    MSG     msg;


    hWnd = Init( hInst, hPrevInst, nCmdShow );
    if( !hWnd )
      return 0;

    while( GetMessage( &msg, 0, 0, 0 ) )
    {
      TranslateMessage( &msg );
```

(continued)

Figure 5-6. *continued*

```
        DispatchMessage( &msg );
    }

    return msg.wParam;
}

/************************************************************************
*                                                                       *
* Init                                                                  *
*                                                                       *
************************************************************************/

static HWND Init( HANDLE hInst, HANDLE hPrevInst, int nCmdShow )
{
    WNDCLASS    wc;
    HWND        hWnd;

    if( !hPrevInst )
    {
      /* register the top-level window class */
      wc.lpszClassName  = szTopLevelClass;
      wc.hInstance      = hInst;
      wc.lpfnWndProc    = TopLevelWndFn;
      wc.hCursor        = LoadCursor( 0, IDC_ARROW );
      wc.hIcon          = LoadIcon( hInst, "TopLevelIcon" );
      wc.lpszMenuName   = NULL;
      wc.hbrBackground  = COLOR_WINDOW+1;
      wc.style          = CS_HREDRAW | CS_VREDRAW;
      wc.cbClsExtra     = 0;
      wc.cbWndExtra     = DLGWINDOWEXTRA;

      if( !RegisterClass( &wc ) )
        return 0;                       /* return 0 if unsuccessful */
    }

    /* save the instance handle */
    hInstance = hInst;

    /* create and display a top-level window
       and several child controls */
    hWnd = CreateDialog( hInst, szTopLevelClass, 0, NULL );

    ShowWindow( hWnd, nCmdShow );

    return hWnd;
}
```

(continued)

Figure 5-6. *continued*

```
/*****************************************************************************
*                                                                            *
* TopLevelWndFn                                                              *
*                                                                            *
*****************************************************************************/

LONG PASCAL FAR
TopLevelWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bDWP = FALSE;
    int     n;

    switch( wMsg )
    {
      case WM_SETFOCUS:
        SetFocus( GetDlgItem( hWnd, IDKEYTRAP ) );
        break;

      case WM_COMMAND:
        if( IDKEYTRAP == wParam )
        {
          if( IsDlgButtonChecked( hWnd, IDKEYTRAP ) )
            for( n=IDCTLFIRST; n<=IDCTLLAST; n++ )
              InstallKeyTrap( GetDlgItem( hWnd, n ) );

          else
            for( n=IDCTLFIRST; n<=IDCTLLAST; n++ )
              UninstallKeyTrap( GetDlgItem( hWnd, n ) );
        }
        break;

      case WM_DESTROY:
        PostQuitMessage( 0 );
        break;

      default:
        bDWP = TRUE;
        break;
    }

    if( bDWP )
      lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

    return lRVal;
}
```

(continued)

Figure 5-6. *continued*

```
/************************************************************************
*                                                                       *
* KeyTrapWndFn                                                          *
*                                                                       *
************************************************************************/

LONG PASCAL FAR
KeyTrapWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LOCALHANDLE hFNStruc;
    NPFNSTRUC   pFNStruc;
    FARPROC     pWndFn;

    /* process the F1 key */
    if( (WM_KEYDOWN == wMsg) && (VK_F1 == wParam) )
      MessageBox( hWnd, "You pressed the F1 key", szAppTitle, MB_OK );

    /* get a pointer to the default window function */
    hFNStruc = GetProp( hWnd, szFNStruc );
    pFNStruc = (NPFNSTRUC)LocalLock( hFNStruc );
    pWndFn = pFNStruc->pDefWndFn;
    LocalUnlock( hFNStruc );

    /* call the default window function */
    return CallWindowProc( pWndFn, hWnd, wMsg, wParam, lParam );
}


/************************************************************************
*                                                                       *
* InstallKeyTrap                                                        *
*                                                                       *
************************************************************************/

static void InstallKeyTrap( HWND hWnd )
{
    LOCALHANDLE hFNStruc;
    NPFNSTRUC   pFNStruc;

    /* allocate storage for pointers to the window functions */
    hFNStruc = LocalAlloc( LHND, sizeof(FNSTRUC) );
    pFNStruc = (NPFNSTRUC)LocalLock( hFNStruc );

    /* save the window-function pointers */
    pFNStruc->pThunk = MakeProcInstance( (FARPROC)KeyTrapWndFn, hInstance );
    pFNStruc->pDefWndFn =
        (FARPROC)SetWindowLong( hWnd, GWL_WNDPROC, (LONG)pFNStruc->pThunk );
```

(continued)

Figure 5-6. *continued*

```
    LocalUnlock( hFNStruc );

    /* save the handle to the pointer data structure */
    SetProp( hWnd, szFNStruc, hFNStruc );
}


/************************************************************************
*                                                                       *
* UninstallKeyTrap                                                      *
*                                                                       *
************************************************************************/

static void UninstallKeyTrap( HWND hWnd )
{
    LOCALHANDLE hFNStruc;
    NPFNSTRUC   pFNStruc;


    /* point to the pointer data structure */
    hFNStruc = RemoveProp( hWnd, szFNStruc );
    pFNStruc = (NPFNSTRUC)LocalLock( hFNStruc );

    /* restore the default window-function pointer */
    SetWindowLong( hWnd, GWL_WNDPROC, (LONG)pFNStruc->pDefWndFn );
    FreeProcInstance( pFNStruc->pThunk );

    /* discard the data structure */
    LocalUnlock( hFNStruc );
    LocalFree( hFNStruc );
}
```

```
/************************************************************************
*                                                                       *
* KEYTRAP.RC resource script                                            *
*                                                                       *
************************************************************************/

#include    <windows.h>
#include    "keytrap.h"

/* icons */
TopLevelIcon ICON keytrap.ico
```

(continued)

Figure 5-6. *continued*

```
KeyTrap DIALOG PRELOAD MOVEABLE DISCARDABLE 42, 32, 236, 54
CAPTION "Key Trap"
CLASS KEYTRAPCLASSNAME
STYLE DS_ABSALIGN | WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
{
  CONTROL "Key Trap", IDKEYTRAP, "Button", BS_AUTOCHECKBOX | WS_TABSTOP |
    WS_CHILD, 4, 20, 42, 12
  CONTROL "", 0, "Static", SS_BLACKRECT | WS_CHILD, 48, 0, 1, 54
  CONTROL "ListBox", 0, "Static", SS_CENTER | WS_CHILD, 52, 2, 66, 8
  CONTROL "", IDLISTBOX, "ListBox", LBS_SORT | WS_BORDER | WS_VSCROLL |
    WS_TABSTOP | WS_CHILD, 52, 12, 66, 33
  CONTROL "Edit", 0, "Static", SS_CENTER | WS_CHILD, 122, 2, 66, 8
  CONTROL "", IDEDIT, "Edit", ES_MULTILINE | WS_BORDER | WS_TABSTOP |
    WS_CHILD, 122, 12, 66, 33
  CONTROL "Button", IDBUTTON, "Button", BS_PUSHBUTTON | WS_TABSTOP |
    WS_CHILD, 192, 12, 38, 33
}
```

```
/************************************************************************
*                                                                       *
* KEYTRAP.H                                                             *
*                                                                       *
************************************************************************/

#define KEYTRAPCLASSNAME    "KeyTrap"

#define IDCTLFIRST          100
#define IDKEYTRAP           (IDCTLFIRST)
#define IDLISTBOX           (IDCTLFIRST+1)
#define IDEDIT              (IDCTLFIRST+2)
#define IDBUTTON            (IDCTLFIRST+3)
#define IDCTLLAST           (IDCTLFIRST+3)
```

```
;************************************************************************
;                                                                       *
; KEYTRAP.DEF module-definition file                                    *
;                                                                       *
;************************************************************************

NAME            KEYTRAP
DESCRIPTION     'KEYTRAP.EXE version 1.0'
```

(continued)

Figure 5-6. *continued*

```
EXETYPE         WINDOWS
STUB            'WINSTUB.EXE'

CODE            LOADONCALL MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE MULTIPLE

SEGMENTS        _TEXT   PRELOAD MOVEABLE DISCARDABLE

HEAPSIZE        512
STACKSIZE       5120

EXPORTS         TopLevelWndFn
                KeyTrapWndFn
```

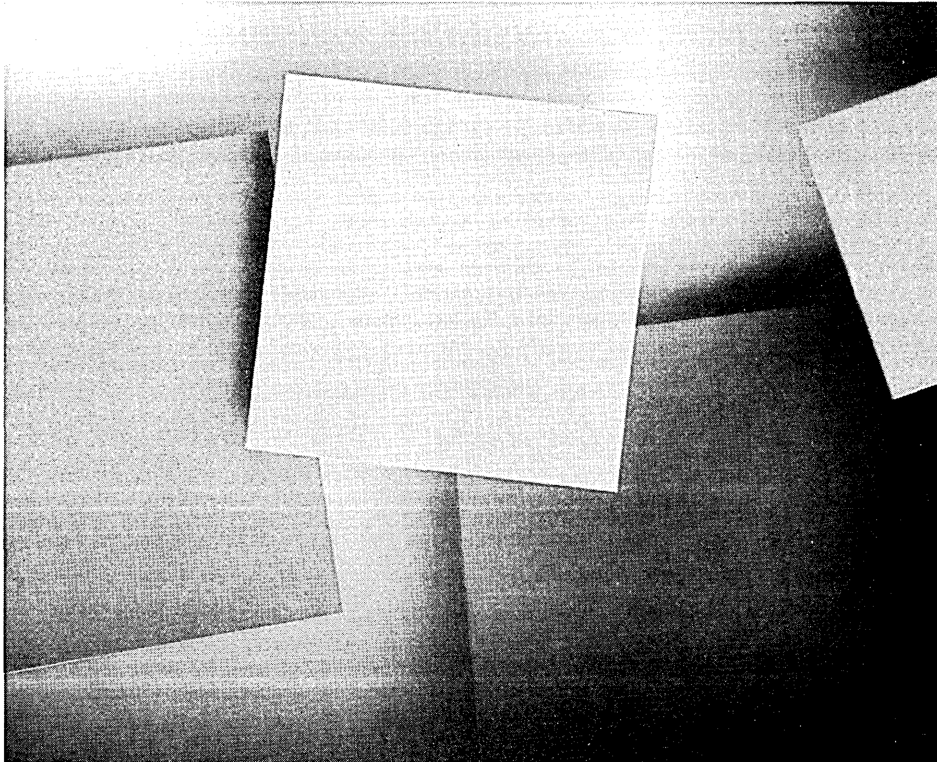
The KEYTRAP application uses message filtering to trap the WM_KEYDOWN message associated with a particular keystroke. When you call *InstallKeyTrap* with the parent window's handle, *KeyTrapWndFn* becomes a message-filter function for all four child-window input controls in the application. *InstallKeyTrap* saves the previous window-function address in the local heap and adds the local-memory handle to the window's property list.

KeyTrapWndFn passes messages up the hierarchy by using *GetProp* to locate the address of the appropriate window function and then calling *CallWindowProc.* The only message that *KeyTrapWndFn* filters is the WM_KEYDOWN message that represents the F1 key. In response to this message, *KeyTrapWndFn* displays a message box. Filtering continues until *UninstallKeyTrap* executes. *UninstallKeyTrap* restores the previous window-function address and cleans up the window's property list.

These examples show how having an object-oriented point of view can add useful generality to your Windows source code. The next step is to embody the object-oriented aspects of the Windows environment in an object-oriented programming language in which objects and classes are easier to manipulate than in a procedural language such as C or Pascal. You might want to explore one of the object-oriented programming languages available for Windows programming, such as Smalltalk, Actor, or C++. Even in C, however, you can improve the design of your Windows applications by taking advantage of object-oriented design elements in Windows.

6

Dynamic Data Exchange (DDE)



Windows is a multitasking environment in which several programs can execute concurrently. It is only natural for concurrent Windows programs to share data with each other. Windows users realize this intuitively by using the clipboard to transfer data among applications.

Although the clipboard is an excellent tool for user-initiated data transfers, its design is not well suited to direct interprocess communication in which Windows applications share data without user intervention. This is where Dynamic Data Exchange (DDE) plays its role. DDE allows Windows applications to communicate directly to share both data and computational tasks.

In the original DDE specification, introduced in version 2 of Windows, DDE is implemented through a set of Windows messages and data structures defined in a C-language include file, DDE.H, in the Windows SDK. The printed protocol for using DDE messages is also part of the Windows SDK. In 1991, Microsoft released the DDE Management Library (DDEML). The DDEML supports a set of API functions that manage DDE communications at a higher level of abstraction than the message-based DDE protocol.

The DDEML was actually implemented using the message-based DDE protocol, so existing Windows applications that use DDE messages are compatibile with DDEML-based applications. However, you should use the DDEML rather than message-based DDE in new Windows applications. The DDEML API is superior because it hides the details of DDE message processing and because it offers additional functionality beyond the capabilities of the message-based DDE protocol. The following discussions of DDE's transaction-based communication model and of the message-based DDE protocol should help you appreciate the overall design of DDE-based interprocess communication. However, you should move on to the subsequent discussion of the DDEML when you design your DDE applications.

Conversations and Transactions

DDE applications share data by means of conversations. A DDE conversation is a logical connection between two different applications, in which the two applications alternately transmit data to each other. A Windows application can support multiple DDE conversations, so any Windows application can exchange data with several other applications at the same time.

Each DDE conversation is structured as a series of transactions between a client and a server. Each transaction consists of a request for data or services and a

corresponding response. The difference between a DDE client and a DDE server lies in the kinds of transactions that each can initiate. A DDE client can initiate any of the following transactions:

- Enumerate DDE services and topics.
- Establish a conversation with a server.
- Request a data item from a server.
- Establish a data link with a server.
- Terminate a data link.
- Send a data item to a server.
- Request a server to execute one or more commands.
- Terminate a conversation.

Only two transactions can be initiated by a DDE server:

- Send a data item to a client.
- Terminate a conversation.

Because a DDE conversation is always initiated by a client, a DDE server must be executing before a client attempts to initiate the conversation.

Message-Based DDE

In message-based DDE, both the client and the server in a DDE conversation are windows. An application that supports DDE creates a window for each DDE conversation in which it participates. Each DDE window can function either as a client or as a server, so a single Windows application can support multiple DDE server and client conversations. An application creates a DDE window each time it begins a new conversation and destroys the window when the conversation terminates. During its lifetime, the DDE window's primary responsibility is to process DDE messages, either as a DDE client or as a DDE server.

Two windows carry out a DDE conversation by exchanging a series of predefined Windows messages, which are shown in Figure 6-1. The messages WM_DDE_INITIATE, WM_DDE_ACK, and WM_DDE_TERMINATE permit handshaking between client and server windows in a DDE conversation so that the windows can exchange messages in an orderly, synchronized manner. The WM_DDE_REQUEST, WM_DDE_ADVISE, and WM_DDE_UNADVISE messages control when data is transferred between windows, and the WM_DDE_DATA

and WM_DDE_POKE messages accompany the data itself. Finally, the WM-_DDE_EXECUTE message allows an application to execute commands or perform some other service on behalf of another application.

Message	***Description***	***Parameters***
Initiating and terminating a conversation		
WM_DDE_INITIATE	Initiate a DDE conversation	HIWORD*(lParam): aTopic* LOWORD*(lParam): aService*
WM_DDE_TERMINATE	Terminate a DDE conversation	
Acknowledging receipt of a DDE message		
WM_DDE_ACK	Acknowledge a DDE message	See Figure 6-2 on page 163.
Data control		
WM_DDE_REQUEST	Request a one-time data transfer	HIWORD*(lParam): aItem* LOWORD*(lParam): cfFormat*
WM_DDE_ADVISE	Request a data link	HIWORD*(lParam): aItem* LOWORD*(lParam): hDDEADVISE*
WM_DDE_UNADVISE	Terminate a data link	HIWORD*(lParam): aItem* LOWORD*(lParam): cfFormat*
Data transfer		
WM_DDE_DATA	Transfer data from server to client	HIWORD*(lParam): aItem* LOWORD*(lParam): hDDEDATA*
WM_DDE_POKE	Transfer data from client to server	HIWORD*(lParam): aItem* LOWORD*(lParam): hDDEPOKE*
Executing commands		
WM_DDE_EXECUTE	Request server to execute a command	HIWORD*(lParam): hCommandString*

Figure 6-1.
Windows DDE messages. Parameters are packed into the high-order and low-order words of lParam. *Parameter names starting with the letter* a *represent atoms; parameter names starting with the letter* h *represent global memory handles. DDEADVISE, DDEPOKE, and DDEDATA are data structures defined in the include file DDE.H. (See also Figure 6-3 on page 166.)*

The message-based DDE specification requires most DDE messages to be transmitted by using *PostMessage*. *SendMessage* is used only for WM_DDE_INITIATE and for WM_DDE_ACK messages sent in response to WM_DDE_INITIATE. The *wParam* parameter of *PostMessage* and *SendMessage* always contains the window handle of the message sender. The high-order and low-order words of the *lParam* parameter contain values whose meanings are different for each DDE message.

To put the DDE messages into perspective, consider how they are used in the context of a DDE conversation. The following overview groups the DDE messages according to the way they are used in transactions: initiating and terminating a conversation, acknowledging receipt of a message, data control, data transfer, and executing commands.

Initiating and Terminating a Conversation

The WM_DDE_INITIATE message is sent by a DDE client to all potential DDE servers. The client broadcasts the message to all overlapped and pop-up windows in the Windows system by calling *SendMessage* with a destination window handle of FFFFH:

```
SendMessage( 0xFFFF, WM_DDE_INITIATE, hClientWnd, lParam );
```

Potential servers reply to the client by calling the *SendMessage* function with the WM_DDE_ACK message. The *lParam* parameter in the WM_DDE_ACK message contains two global atoms that identify the server and a topic of conversation.

The WM_DDE_TERMINATE message can be sent by either partner in a DDE conversation to terminate the conversation. The *lParam* parameter is not used in this message.

Acknowledging Receipt of a DDE Message

The WM_DDE_ACK message is used to acknowledge a variety of DDE messages at different times in a DDE conversation. The content of the *lParam* parameter for WM_DDE_ACK depends on which message is being acknowledged, as shown in Figure 6-2.

Message Acknowledged	Parameters	Notes
WM_DDE_INITIATE	HIWORD(*lParam*): *aTopic* LOWORD(*lParam*): *aService*	Sent by server to client. Informs client of server's support for specified topic.
WM_DDE_DATA	HIWORD(*lParam*): *aItem* LOWORD(*lParam*): *wStatus*	Sent by client to server. Acknowledges receipt of data. Used only when explicitly requested by server. (See *fAckReq* bit in DDEDATA, Figure 6-3.)
WM_DDE_POKE	HIWORD(*lParam*): *aItem* LOWORD(*lParam*): *wStatus*	Sent by server to client. Acknowledges receipt of data.
WM_DDE_EXECUTE	HIWORD(*lParam*): *hCommands* LOWORD(*lParam*): *wStatus*	Sent by server to client. Acknowledges execution of a command string.
WM_DDE_REQUEST	HIWORD(*lParam*): *aItem* LOWORD(*lParam*): *wStatus*	Sent by server to client. Used only for negative acknowledgment of request for data.
WM_DDE_ADVISE	HIWORD(*lParam*): *aItem* LOWORD(*lParam*): *wStatus*	Sent by server to client. Acknowledges initiation of a data link.
WM_DDE_UNADVISE	HIWORD(*lParam*): *aItem* LOWORD(*lParam*): *wStatus*	Sent by server to client. Acknowledges termination of a data link.

Figure 6-2.
Use of the WM_DDE_ACK message to acknowledge DDE messages.

Data Control

The WM_DDE_REQUEST message can be posted by a client to request that a server transmit a particular data item. The *lParam* parameter describes the data item by name and specifies a data format. The server responds to a WM_DDE_REQUEST message by posting a WM_DDE_DATA message containing the requested data.

The WM_DDE_ADVISE message sets up a data link between server and client. By using a DDE data link, a server can send data items to a client without requiring the client to request each data item explicitly. The *lParam* parameter of WM_DDE_ADVISE describes the name and format of a data item; it also describes the handshaking method to be used for individual data transfers within the data link.

A client initiates a data link by posting WM_DDE_ADVISE to a server. The server acknowledges the data link by posting a WM_DDE_ACK message to the client. The server can then post unsolicited data to the client. The data link continues until the client terminates it by posting WM_DDE_UNADVISE or until the conversation is terminated with WM_DDE_TERMINATE.

Data Transfer

The WM_DDE_DATA and WM_DDE_POKE messages are used to transfer data between DDE windows. The WM_DDE_DATA message is used to transfer data from a server to a client, either in response to an explicit WM_DDE_REQUEST message from the client or as part of an ongoing data link. WM_DDE_POKE is used to transfer data from a client to a server. The *lParam* parameter in both messages identifies the data with a global atom and also contains a handle to a global memory object that contains the data being transferred.

Executing Commands

A client uses the WM_DDE_EXECUTE message to transmit commands to a server. To use WM_DDE_EXECUTE, a client formats a character string that contains one or more commands and passes a reference to the character string in the WM_DDE_EXECUTE *lParam* parameter. The server parses the command string, executes the commands, and posts WM_DDE_ACK to the client to acknowledge that the commands were processed.

Service, Topic, and Item Names

The DDE specification uses a three-level hierarchy to describe the data content of a DDE conversation. This descriptive hierarchy consists of a *service* name, a *topic* of conversation, and the name of a particular *item* of data. When a DDE conversation is initiated, the server and client must agree on both the name of the service and the topic of conversation. After the conversation is established, specific data values can be referenced by name and transferred between the client and the server.

The following conversation is a not-quite-realistic analogy of how this descriptive hierarchy works. In this example, a client initiates a conversation with a server by specifying *Customer Support* (service) on the *Windows SDK* (topic) and by requesting a *price* (item of data):

> **Client** (*sends WM_DDE_INITIATE*): Hello, is this *Customer Support* (service name)? I'd like to initiate a conversation about the *Windows SDK* (topic name).
>
> **Server** (*sends WM_DDE_ACK*): Yes, this is *Customer Support* (service name). I will be happy to converse with you about the *Windows SDK* (topic name).
>
> **Client** (*posts WM_DDE_REQUEST*): What is the *price* (item name) of the Windows SDK?
>
> **Server** (*posts WM_DDE_DATA*): The *price* (item name) is $19.95.
>
> **Client** (*posts WM_DDE_TERMINATE*): OK, goodbye.
>
> **Server** (*posts WM_DDE_TERMINATE*): Goodbye.

A DDE server uses the three-level naming hierarchy to describe the context in which it provides data to DDE clients. A server application can support one or more service names, each of which supports multiple topics. The server makes different sets of data items available to clients in the context of each service-topic combination it supports.

A DDE client exploits this three-level descriptive scheme to discriminate among potential data servers in order to obtain a particular item of data. A client can select a particular service by name, or it can enumerate multiple services that support a particular topic and then choose among them to initiate a conversation. In either case, it can then request specific data items by name and format.

The hierarchical naming strategy is powerful because a client can identify data using only descriptive names. The DDE naming hierarchy hides the details of a server's implementation from a client. The client needs no special knowledge about a server's implementation, whether the server is reading disk files, transferring data across a network or a remote communications link, or calculating data on the fly.

Traditionally, a Windows application that acts as a DDE server uses its application name (module name) as a DDE service name. For example, the Windows Program Manager application, PROGMAN.EXE, supports the DDE service name *ProgMan*. However, a single Windows application can support multiple DDE service names, each of which may differ from the application name, and each of

which may cover a different gamut of topics. Moreover, an application can change the services and topics it supports in response to changing circumstances. For example, a DDE application that performs remote communications might support a particular service name only when a link to a remote computer is active.

Working with DDE Data

In DDE, data values are exchanged by storing them in shared global memory. DDE messages use two kinds of global data: shareable global-memory blocks and global atoms. Shareable global-memory blocks provide the means of transferring data between applications. Global atoms represent service, topic, and data-item names. All DDE messages except WM_DDE_TERMINATE use atom handles and global-memory handles as parameters by packing them into the low-order and high-order words of *lParam*.

Shareable Global Memory

Every global-memory block associated with a DDE message is formatted with one of three predefined data structures: DDEADVISE, DDEPOKE, or DDEDATA. These three data structures are shown in Figure 6-3. Each of the data structures starts with a 16-bit word of flag bits followed by a 16-bit integer, *cfFormat*, which contains a clipboard-format value that describes the format of the shared data. In DDEDATA and DDEPOKE, the data structures used with WM_DDE_DATA and WM_DDE_POKE, these two 16-bit values are followed by actual data.

```
/* DDEADVISE: used with WM_DDE_ADVISE */
typedef struct
{
  unsigned  reserved:   14,          /* bits 0-13 */
            fDeferUpd:   1,          /* bit 14 */
            fAckReq:     1;          /* bit 15 */
  int       cfFormat;
}
                DDEADVISE;

/* DDEPOKE: used with WM_DDE_POKE */
typedef struct
{
  unsigned  unused:     13,         /* bits 0-12 */
```

Figure 6-3. *(continued)*

Data structures used with DDE messages. These data structures are defined in DDE.H in the Windows SDK.

Figure 6-3. *continued*

```
            fRelease:    1,          /* bit 13 */
            fReserved:   2;          /* bits 14-15 */
  int       cfFormat;
  BYTE      Value[1];
}
                DDEPOKE;

/* DDEDATA: used with WM_DDE_DATA */
typedef struct
{
  unsigned  unused:      12,         /* bits 0-11 */
            fResponse:   1,          /* bit 12 */
            fRelease:    1,          /* bit 13 */
            reserved:    1,          /* bit 14 */
            fAckReq:     1;          /* bit 15 */
  int       cfFormat;
  BYTE      Value[1];
}
                DDEDATA;
```

The DDE specification requires that the global memory allocated for DDEDATA and DDEPOKE data structures be shareable. Shareable global-memory blocks are allocated using the GMEM_DDESHARE flag in the call to *GlobalAlloc*:

```
hMem = GlobalAlloc( GHND | GMEM_DDESHARE,
                    dwDataSize + sizeof(DDEDATA) );
```

GDI Objects

In addition to shareable global-memory blocks, you can also share GDI objects in a DDE conversation. Do this by using handles to GDI objects as data in DDEDATA or DDEPOKE data structures. For example, a DDE server can share a GDI bitmap by passing the handle returned by *CreateBitmap* or *CreateCompatibleBitmap* as data in the *Value* array of the DDEDATA data structure in a WM_DDE_DATA message.

Global Atoms

In message-based DDE, global atoms generally reference plain-text strings that represent service, topic, and item names. You might find it convenient, however, to use integer strings instead of text strings, particularly if your application references a large number of different data items. (An integer string is an integer formatted as an ASCII string preceded by the # character. For example, *#32666* is the integer-string representation of the integer 32666.) Also, don't forget that atoms are case-insensitive: *System* and *SYSTEM* represent the same atom.

In general, the sender of a DDE message creates the atoms associated with the message with a call to *GlobalAddAtom.* The recipient uses *GlobalFindAtom* or *GlobalGetAtomName* to identify the atoms associated with the message. The recipient can then delete the atom by calling *GlobalDeleteAtom,* or it can reuse the same atom if it needs to post a message in reply.

There are two exceptions to this rule. If *PostMessage* fails to post a DDE message, the atoms associated with the message should be deleted. Also, in the case of the atoms used in a WM_DDE_INITIATE message, the DDE client that broadcasts WM_DDE_INITIATE must be the one to delete the atoms it creates. A DDE server that receives WM_DDE_INITIATE must not delete or reuse the atoms associated with the message. This makes sense because more than one server may respond to a single WM_DDE_INITIATE broadcast. If a server deleted the atoms, another server that receives the broadcast message would find that the associated atoms were invalid.

Flags

The message-based DDE protocol uses a number of flags to control the flow of DDE messages, to indicate the disposition of globally shared objects, and to indicate the status of various transactions. These flags appear in the predefined DDE data structures DDEADVISE, DDEPOKE, and DDEDATA. There are also flags in the status word associated with the WM_DDE_ACK message.

Flags for message control

The flag word in the DDE data structures DDEDATA and DDEADVISE lets you fine-tune two DDE message transactions. In the case of WM_DDE_DATA, the *fAckReq* bit in the DDEDATA data structure indicates whether a client should acknowledge a WM_DDE_DATA message by posting WM_DDE_ACK to the server. If the *fAckReq* bit is 1, the client should post WM_DDE_ACK to the server; if the *fAckReq* bit is 0, no acknowledgment is necessary. By setting *fAckReq* to 1, a server can ensure that a client successfully processes WM_DDE_DATA messages in the order they are received.

In the case of WM_DDE_ADVISE, both the *fAckReq* and the *fDeferUpd* bits in the DDEADVISE data structure affect subsequent WM_DDE_DATA messages sent by the server through a data link. The *fAckReq* bit specifies whether the *fAckReq* bit in subsequent WM_DDE_DATA messages should be set and thus whether the client will be expected to acknowledge unsolicited WM_DDE-_DATA messages it receives through the data link. A DDE server that supports a

data link copies the *fAckReq* value from the DDEADVISE data structure into each WM_DDE_DATA message it posts through the data link to the client.

The *fDeferUpd* bit in the DDEADVISE data structure specifies whether the server will include data in the WM_DDE_DATA messages it sends through a DDE data link. If *fDeferUpd* is 0, the server includes the global memory handle of a DDEDATA data structure in each WM_DDE_DATA message. If *fDeferUpd* is 1, the server passes a null value instead of a global memory handle with each WM_DDE_DATA message. The null WM_DDE_DATA message serves as an alarm that notifies the client that the server has changed a data item's value. It is then up to the client either to request the data by posting WM_DDE_REQUEST or to ignore the WM_DDE_DATA message.

Flags for global memory

The *fRelease* bit in each DDEDATA and DDEPOKE data structure determines how to manage the block of global memory that contains the data structure. When the *fRelease* bit is set to 1, the recipient of the WM_DDE_DATA or WM_DDE_POKE message should free the memory block after it has finished using the block. When the *fRelease* bit is 0, the sender of the message remains responsible for freeing the memory block; in this case, the recipient should not modify the data within the memory block because the sender might reuse the same block of memory in a subsequent data transmission. If you use this technique, be sure to set the *fAckReq* bit to 1 so that the recipient of the WM_DDE_DATA or WM_DDE_POKE message will post a WM_DDE_ACK message that indicates when it is safe for the sender to reuse the memory object.

There is a subtle trap in this otherwise commonsense memory-management strategy. The problem potentially can occur whenever an application sets the *fRelease* bit to 1 when it posts a WM_DDE_DATA or WM_DDE_POKE message. If the sending application terminates before the recipient of the message can access the memory handle, Windows' memory manager invalidates the handle and frees the memory. The recipient will then be in error when it attempts to access the already-freed handle.

The solution to this problem is simple. When the *fRelease* bit is set to 1, the recipient should assume ownership of the memory block by calling *GlobalReAlloc*:

```
hMem = GlobalReAlloc( hMem, 0L, GMEM_MODIFY | GMEM_DDESHARE );
```

Then the memory block will remain allocated even if the sending application terminates.

Status flags

The message-based DDE specification includes flags that can be used to report the status of DDE transactions. The *fResponse* flag in the DDEDATA data structure indicates whether a WM_DDE_DATA message was posted in response to an explicit WM_DDE_REQUEST (*fResponse*=1) or as part of an active data link (*fResponse*=0).

The WM_DDE_ACK message also uses status flags, but these are returned in a single word, formatted as a DDEACK data structure, in the low-order word of the *lParam* parameter of the message, as shown in Figure 6-4. The status word contains two 1-bit flags, *fAck* and *fBusy*, as well as an 8-bit, application-specific return value. Both the flags and the status value should be carefully managed in any DDE program.

Bit	Name	Description
15	*fAck*	1=Positive acknowledgment 0=Negative acknowledgment
14	*fBusy*	1=Busy 0=Not busy
8-13		(reserved)
0-7	*bAppReturnCode*	Application-specific return value

Figure 6-4.
The wStatus *word in WM_DDE_ACK. This word is defined as a DDEACK data structure in DDE.H in the Windows SDK.*

The *fAck* flag indicates whether the associated WM_DDE_ACK message represents a positive or a negative acknowledgment of a previous DDE message. For example, when a server posts a WM_DDE_ACK message in response to a WM_DDE_POKE message from a client, the server sets the *fAck* bit to 1 to indicate that it successfully accepted the data associated with the message; it sets *fAck* to 0 to indicate that the data was not processed.

If the sender of a WM_DDE_ACK message sets the *fAck* bit to 0, it has the option of setting the *fBusy* bit as well. The *fBusy* bit indicates that the sender was temporarily unable to process a previous DDE message. In effect, setting the *fBusy* bit implies "Try again later." Of course, if *fAck* is set to 1, *fBusy* must be 0.

Proper use of the *fBusy* bit is important because the DDE specification requires that a DDE window process all DDE messages it receives in the order in which

they are received. This becomes an issue when an application performs some prolonged computational or communications activity in response to a DDE message. If the application is too busy to process subsequent DDE messages, it should respond to subsequent DDE messages by posting WM_DDE_ACK with *fBusy* set to 1.

Data Formats

Because the message-based DDE protocol provides a consistent mechanism for specifying the format of shared data, cooperating DDE applications can agree on the data format for each data transfer. For example, when a DDE client requests a data item in a WM_DDE_REQUEST message, it can specify a preferred data format in the *cfFormat* parameter in the low-order word of *lParam.* If the server cannot support the requested data format, it will refuse the request by returning WM_DDE_ACK with the *fAck* bit (in the return status word) set to 0. The client can subsequently request alternative data formats until it finds one that the server can support.

The value you specify in *cfFormat* must be a valid clipboard data format. If your data does not conform to one of the predefined clipboard data formats listed in Figure 6-5, both server and client should call *RegisterClipboardFormat* to register a clipboard format ID that can be used in subsequent DDE messages.

Format	***Description***
CF_TEXT	Null-terminated ASCII string
CF_BITMAP	Handle to a bitmap (defined by BITMAP data structure in WINDOWS.H)
CF_METAFILEPICT	Metafile picture (defined by METAFILEPICT data structure in WINDOWS.H)
CF_SYLK	Microsoft Symbolic Link format
CF_DIF	Software Arts' Data Interchange Format
CF_TIFF	Tagged Image File Format
CF_OEMTEXT	Same as CF_TEXT but using OEM character set
CF_DIB	Device-independent bitmap (defined by BITMAPINFO data structure in WINDOWS.H)

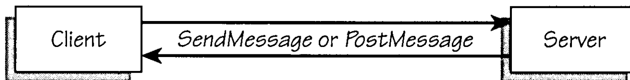
Figure 6-5.
Clipboard data formats defined in WINDOWS.H.

The DDE Management Library

The traditional way to use DDE in a Windows application is to embed a set of DDE message-handling functions in the application. This is the approach adopted in many well-known Windows applications, including the original Microsoft Excel and Word for Windows. When you consider the amount of detail in the message-based DDE protocol, however, it becomes clear that the source code required to support DDE processing is better encapsulated in a dynamic link library.

This is exactly the purpose of the DDEML, the DDE Management Library. The heart of the DDEML is a dynamic link library (DDEML.DLL) that relieves applications of the burden of processing individual DDE messages. Unlike message-based DDE, in which applications communicate directly with one another, DDEML-aware applications communicate with only the DDEML, as shown in Figure 6-6. An application calls a set of DDEML API functions to carry out DDE transactions.

Message-based DDE:



DDEML:

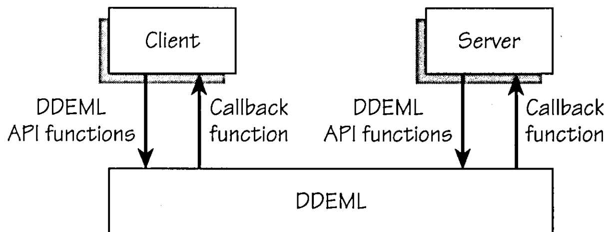


Figure 6-6.
In message-based DDE (top), a DDE client and server communicate directly, using SendMessage *and* PostMessage. *With the DDEML (bottom), all DDE transactions are performed through calls from applications to DDEML API functions and through calls from the DDEML to a callback function in each application.*

A program that uses the DDEML must include an exported callback function that the DDEML can call to notify the program when DDE transactions occur. Like a window function, a DDEML callback function is an exported far function with a predefined set of parameters, including a transaction-type identifer that can be used to select transaction-specific actions, as shown in Figure 6-7.

```
HDDEDATA EXPENTRY
DdeCallback( WORD      wType,        /* transaction ID */
           , WORD      wFmt,         /* clipboard data format */
             HCONV     hConv,        /* conversation handle */
             HSZ       hsz1,         /* string handle 1 */
             HSZ       hsz2,         /* string handle 2 */
             HDDEDATA hDDEData,      /* global data handle */
             DWORD     dwData1,      /* 32-bit data */
             DWORD     dwData2 );    /* 32-bit data */
```

Figure 6-7.
A function prototype for an application-defined DDEML callback function.

Both DDEML API functions and application-defined callback functions use data types and data structures that are designed to facilitate DDE transaction management, as shown in Figure 6-8. These data types and structures are defined in a C-language include file, DDEML.H, which Microsoft distributes along with the DDEML dynamic link library. The DDEML uses handles to identify DDE conversations (data type HCONV), lists of conversations (HCONVLIST), strings that represent service, topic, or item names (HSZ), and shared blocks of global memory (HDDEDATA).

```
/* EXPENTRY is used in declaring a DDEML callback function. */
#define EXPENTRY   _export far pascal

typedef DWORD HSZ;            /* string handle */
typedef DWORD HDDEDATA;       /* global data handle */
typedef DWORD HCONV;          /* conversation handle */
typedef DWORD HCONVLIST;      /* list of conversation
                                  handles */

typedef struct
{
  HSZ        hszSvc;          /* service name */
  HSZ        hszTopic;        /* topic name */
}
        HSZPAIR;              /* pair of string handles */
```

Figure 6-8. *(continued)*
Some of the data types and data structures defined in DDEML.H.

Figure 6-8. *continued*

```
typedef struct
{
  WORD        cb;             /* size of this data structure */
  WORD        wFlags;         /* (reserved for future use) */
  WORD        wCountryID;     /* country code for topic and
                                 item strings */
  int         iCodePage;      /* code page for topic and
                                 item strings */
  DWORD       dwLangID;       /* language ID for topic and
                                 item strings */
  DWORD       dwSecurity;     /* private security code */
}
        CONVCONTEXT;


typedef struct
{
  DWORD       cb;             /* size of this data structure */
  DWORD       hUser;          /* user-defined data */
  HCONV       hConvPartner;   /* hConv for partner */
  HSZ         hszSvcPartner;  /* service name of partner */
  HSZ         hszSvcNameReq;  /* service requested
                                 for connection */
  HSZ         hszTopic;       /* topic for conversation */
  HSZ         hszItem;        /* transaction item name */
  WORD        wFmt;           /* clipboard data format */
  WORD        wType;          /* current transaction ID */
  WORD        wStatus;        /* ST_* conversation
                                 status flags */
  WORD        wConvst;        /* XST_* conversation
                                 status flags */
  WORD        wLastError;     /* last transaction error */
  HCONVLIST   hConvList;      /* link to previous hConvList */
  CONVCONTEXT ConvCtxt;       /* conversation context */
}
        CONVINFO;

typedef CONVINFO FAR *  PCONVINFO;
```

DDEML Transaction Processing

A program initiates a DDE transaction by calling one of the DDEML API functions. The DDEML can then process the transaction either synchronously or asynchronously. When the DDEML processes a synchronous transaction, the transaction completes before the API function call returns. When the DDEML

processes a transaction asynchronously, the API function returns immediately, before the transaction is processed. Later, when the transaction completes, the DDEML notifies the program by calling its callback function.

The ability to process transactions asynchronously is important because it lets a DDE application continue to run while it waits for a prolonged DDE transaction to complete. For example, a DDE client can post an asynchronous request for data to a DDE server and then carry out other actions until the server responds to the data request. Although both synchronous transaction-handling and asynchronous transaction-handling methods are implicit in the original DDE message set, implementing both kinds of transaction processing is a chore that few programmers tackled prior to the appearance of the DDEML.

The DDEML API

The DDEML defines a set of 26 API functions that support both server and client transactions, as shown in Figure 6-9. To ensure compatibility with future versions of Windows, the DDEML API functions frequently use DWORD (32-bit) data values and abstract data types such as handles to strings and global-memory blocks. The function prototypes for all the DDEML functions are in DDEML.H.

Function	***Description***	***Parameters***	***Return Value***
DDEML interface management			
DdeInitialize	Registers a callback function; sets a transaction filter	Pointer to returned instance identifier (LPDWORD) Pointer to callback function (PFNCALLBACK) Command and filter flags (DWORD) Reserved: must be 0 (DWORD)	Result code (WORD)
DdeUninitialize	Terminates all DDEML processing for an application	Instance identifier (DWORD)	TRUE if no error (BOOL)
DdeGetLastError	Returns current DDEML error status	Instance identifier (DWORD)	Error code (WORD)

Figure 6-9. *(continued)*
The 26 DDEML API functions.

Figure 6-9. *continued*

Function	Description	Parameters	Return Value
Conversation management			
DdeNameService	Registers a service name	Instance identifier (DWORD) Service name (HSZ) Reserved: must be 0 (HSZ) Command flags (WORD)	TRUE if no error (BOOL)
DdeConnect	Initiates a DDE conversation	Instance identifier (DWORD) Service name (HSZ) Topic name (HSZ) Pointer to conversation context data (PCONVCONTEXT)	Conversation handle (HCONV)
DdeConnectList	Enumerates DDE services; establishes multiple conversations with DDE servers	Instance identifier (DWORD) Service name (HSZ) Topic name (HSZ) Conversation-list handle (HCONV) Pointer to conversation context data (PCONVCONTEXT)	Conversation-list handle (HCONVLIST)
DdeQueryNextServer	Returns the next conversation handle in a conversation list	Conversation-list handle (HCONVLIST) Previous conversation handle (HCONV)	Next conversation handle (HCONV)
DdeQueryConvInfo	Obtains current status of a DDE conversation	Conversation handle (HCONV) Transaction identifier (DWORD) Pointer to returned status data for the conversation (PCONVINFO)	Number of bytes of status data returned (WORD)
DdeDisconnect	Terminates a DDE conversation	Conversation handle (HCONV)	TRUE if no error (BOOL)
DdeDisconnectList	Terminates multiple conversations	Conversation-list handle (HCONVLIST)	TRUE if no error (BOOL)

(continued)

Figure 6-9. *continued*

Function	***Description***	***Parameters***	***Return Value***
Transaction management			
DdeClientTransaction	Begins a client-initiated transaction	Pointer to shared global data *or* shared global-memory handle (LPBYTE) Data length *or* –1 if global-memory handle specified (DWORD) Conversation handle (HCONV) Item name (HSZ) Clipboard data format (WORD) Transaction type (WORD) Timeout duration (DWORD) Pointer to returned result: synchronous-transaction status flags *or* asynchronous transaction ID (LPDWORD)	Shared global-memory handle *or* status flag (HDDEDATA)
DdeAbandon-Transaction	Aborts an asynchronous transaction	Instance identifier (DWORD) Conversation handle (HCONV) Transaction identifier *or* 0 to abandon all transactions (DWORD)	TRUE if no error (BOOL)
DdeEnableCallback	Blocks (enables) or unblocks (disables) transactions	Instance identifier (DWORD) Conversation handle (HCONV) Enable or disable command (WORD)	TRUE if no error (BOOL)
DdePostAdvise	Updates a data link	Instance identifier (DWORD) Topic name (HSZ) Item name (HSZ)	TRUE if no error (BOOL)

(continued)

Figure 6-9. *continued*

Function	Description	Parameters	Return Value
DdeSetUserHandle	Associates an application-defined value with a conversation and transaction identifier	Conversation handle (HCONV) Transaction identifier (DWORD) Application-defined value (DWORD)	TRUE if no error (BOOL)
String management			
DdeCreateStringHandle	Creates a handle for a specified string	Instance identifier (DWORD) String pointer (LPSTR) Code-page identifier (int)	String handle (HSZ)
DdeQueryString	Obtains string data and length	Instance identifier (DWORD) String handle (HSZ) Pointer to buffer to receive string *or* NULL to obtain string length only (LPSTR) Size of buffer (DWORD) Code-page identifier (int)	Length of the returned string (DWORD)
DdeKeepStringHandle	Increments the usage count for a string handle	Instance identifier (DWORD) String handle (HSZ)	TRUE if no error (BOOL)
DdeFreeStringHandle	Decrements the usage count for a string handle and frees the handle if the count equals 0	Instance identifier (DWORD) String handle (HSZ)	TRUE if no error (BOOL)
DdeCmpStringHandles	Case-insensitive string comparison	String handle 1 (HSZ) String handle 2 (HSZ)	–1: string 1 < string 2; 0: string 1 = string 2; 1: string 1 > string 2

(continued)

Figure 6-9. *continued*

Function	**Description**	**Parameters**	**Return Value**
Memory management			
DdeCreateDataHandle	Creates and initializes a block of shareable global memory	Instance identifier (DWORD) Pointer to buffer containing initial data (LPBYTE) Size of global-memory block (DWORD) Offset of start of initial data (DWORD) Item name (HSZ) Clipboard data format (WORD) Creation flags (WORD)	Global-memory handle (HDDEDATA)
DdeFreeDataHandle	Frees a block of shareable global memory	Global-memory handle (HDDEDATA)	TRUE if no error (BOOL)
DdeAccessData	Obtains a pointer to a block of shared, read-only global memory	Global-memory handle (HDDEDATA) Pointer to returned data length (LPDWORD)	Pointer to global memory block (LPBYTE)
DdeAddData	Copies data into a block of shared global memory	Global-memory handle (HDDEDATA) Pointer to data to be copied (LPBYTE) Data length (DWORD) Offset within the global-memory block (DWORD)	New global-memory handle (HDDEDATA)
DdeGetData	Copies data from a block of shared global memory	Global-memory handle (HDDEDATA) Pointer to buffer to receive copied data *or* NULL to obtain data length only (LPBYTE) Data length (DWORD) Offset within the global-memory block (DWORD)	Data length (DWORD)
DdeUnaccessData	Releases a pointer to a block of shared global memory	Global-memory handle (HDDEDATA)	TRUE if no error (BOOL)

DDEML interface management

Most of the DDEML API functions control an application's DDE conversations. However, three important DDEML functions control an application's interaction with the DDEML itself. These are *DdeInitialize*, *DdeUninitialize*, and *DdeGetLastError*.

An application must call *DdeInitialize* before it uses any other DDEML function. *DdeInitialize* serves two important purposes. First it passes the DDEML a pointer to a program's callback function. The pointer you specify in your call to *DdeInitialize* must be that of an instance thunk (created by *MakeProcInstance*) unless the callback function is defined in a dynamic link library.

Second *DdeInitialize* establishes filters for callback transactions. When you call *DdeInitialize*, you must specify one or more of the flags shown in Figure 6-10. The DDEML will prevent the transactions you specify from reaching the callback function. In general, you should use the APPCLASS_STANDARD flag for a server callback function. For a client callback function, use APPCLASS-_STANDARD | APPCMD_CLIENTONLY so that the callback function receives only transactions relevant to DDE client processing.

The APPCLASS_MONITOR flag allows a DDEML application to monitor DDEML activity. (The DDESPY utility provided with the DDEML is one such application.) However, DDEML monitoring is not as straightforward as it might appear. For example, a DDEML monitor application might not be aware of the DDE activity of message-based, non-DDEML applications. Also, a DDEML monitor application should not also function as a DDEML server or client. Such an application can get into trouble when it attempts to monitor its own DDEML activity. If you want to design a DDEML monitor application, you need to use additional *DdeInitialize* flags and data structures that are defined in DDEML.H and described in the DDEML documentation.

DdeInitialize returns an application-instance identifier, a value that the DDEML uses to associate an instance of a Windows module (application or DLL) with its DDE conversations. Many of the DDEML API functions use the instance identifier as a parameter. You should therefore call *DdeInitialize* before calling any other DDEML function in an application.

The *DdeUninitialize* function terminates DDEML processing for an instance of an application. Before an application terminates, it should call *DdeUninitialize* to ensure that any active DDE conversations are terminated cleanly. In addition to terminating conversations with other DDEML applications, *DdeUninitialize* also posts WM_DDE_TERMINATE to message-based, non-DDEML applications and releases any associated data internal to the DDEML.

Flag	***Transactions Not Sent by DDEML to Callback Function***
APPCLASS_MONITOR	(Used in DDEML monitoring applications)
APPCLASS_STANDARD	XTYP_MONITOR
APPCMD_CLIENTONLY	XTYP_CONNECT, XTYP_WILDCONNECT, XTYP_ADVSTART, XTYP_EXECUTE, XTYP_POKE, XTYP_REQUEST
APPCMD_FILTERINITS	XTYP_CONNECT, XTYP_WILDCONNECT
CBF_FAIL_SELFCONNECTIONS	XTYP_CONNECT and XTYP_WILDCONNECT from the same instance of an application
CBF_FAIL_CONNECTIONS	XTYP_CONNECT and XTYP_WILDCONNECT
CBF_FAIL_ADVISES	XTYP_ADVSTART and XTYP_ADVSTOP (returns DDE_FNOTPROCESSED to the client)
CBF_FAIL_EXECUTES	XTYP_EXECUTE (returns DDE_FNOTPROCESSED to the client)
CBF_FAIL_POKES	XTYP_POKE (returns DDE_FNOTPROCESSED to the client)
CBF_FAIL_REQUESTS	XTYP_REQUEST (returns DDE_FNOTPROCESSED to the client)
CBF_FAIL_ALLSVRXACTIONS	XTYP_CONNECT, XTYP_WILDCONNECT, XTYP_ADVSTART, XTYP_EXECUTE, XTYP_POKE, XTYP_REQUEST (returns DDE_FNOTPROCESSED to the client)
CBF_SKIP_CONNECT_CONFIRMS	XTYP_CONNECT_CONFIRM
CBF_SKIP_REGISTRATIONS	XTYP_REGISTER
CBF_SKIP_UNREGISTRATIONS	XTYP_UNREGISTER
CBF_SKIP_DISCONNECTS	XTYP_DISCONNECT
CBF_SKIP_ALLNOTIFICATIONS	XTYP_CONNECT_CONFIRM, XTYP_REGISTER, XTYP_UNREGISTER, XTYP_DISCONNECT

Figure 6-10.
Flags used with DdeInitialize *to specify transaction filters for a DDEML callback function.*

The *DdeGetLastError* function is a general-purpose function that a program can call after any other DDEML function or transaction indicates that an error has occurred. *DdeGetLastError* returns a value that indicates the nature of the error. (The DDEML.H include file defines the gamut of error values with identifiers that begin with DMLERR_.) The DDEML retains only one error value for calls to *DdeGetLastError*. A program should call *DdeGetLastError* immediately after it detects that an error has occurred because the next DDEML API call will update the error value.

Conversation management

There are seven DDEML API functions you can use to enumerate the services and topics supported by DDE server applications, to initiate and terminate conversations, and to determine the current state of an active conversation. These functions explicitly provide the higher-level support for conversation management that is implicit in the DDE messages WM_DDE_INITIATE, WM_DDE_ACK, and WM_DDE_TERMINATE.

An application uses *DdeNameService* to register and unregister DDE service names. When a server application registers a service name, the DDEML keeps track of the name so that DDE clients can subsequently initiate conversations with it. When the server application no longer supports the service, the server calls *DdeNameService* again to unregister the name. The DDEML notifies other DDEML applications each time a name is registered or unregistered by calling each application's callback function.

A DDE client calls *DdeConnect* and *DdeConnectList* to initiate DDE conversations. Both functions require you to specify a service name and a topic name. You can also use a wildcard specification (a null string handle) for the service name, the topic name, or both. *DdeConnect* establishes a single DDE conversation (even if multiple servers support the service and topic you specify) and returns a handle that identifies the conversation. *DdeConnectList* establishes conversations with all servers that support the service-topic combination you specify and returns a handle to a list of the newly initiated conversations.

DdeConnectList uses its fourth parameter, *hConvList*, to avoid duplicating existing DDE conversations. If you call *DdeConnectList* more than once, you should specify the conversation-list handle returned from the first call to *DdeConnectList* in subsequent calls to the function. *DdeConnectList* will refer to the list of existing conversations and establish a new conversation only if it does not duplicate a conversation already in the list.

A client application can use *DdeQueryNextServer* and *DdeQueryConvInfo* to examine the list of conversations returned by a call to *DdeConnectList*. The conversation list is a linked list of CONVINFO data structures. *DdeQueryNextServer* uses the links to traverse the list and returns a handle to the next conversation. *DdeQueryConvInfo* returns a pointer to a specified conversation's CONVINFO data. Usually, you use *DdeQueryNextServer* and *DdeQueryConvInfo* in a loop such as the following:

```
HCONVLIST hConvList;
HCONV    hConv;
CONVINFO ConvInfo;

/* create or update a conversation list */
hConvList = DdeConnectList( ... );

  :

/* get a handle to the first conversation in the list * /
/* by specifying a null handle to the previous conversation */
hConv = DdeQueryNextServer( hConvList, 0L );

while( hConv )
{
 /* get conversation info for the current hConv */
 DdeQueryConvInfo( hConv, QID_SYNC, &ConvInfo );

 /* examine the contents of the ConvInfo data structure */

  :

 /* get the next conversation handle in the list */
 hConv = DdeQueryNextServer( hConvList, hConv );
}
```

The *DdeDisconnect* and *DdeDisconnectList* functions complement *DdeConnect* and *DdeConnectList*. You use *DdeDisconnect* to terminate a single conversation and *DdeDisconnectList* to terminate all conversations in a list. Both of these functions are intuitive and easy to use, yet there is a common situation in which you don't need them at all: If an application is about to shut down, it can implicitly terminate all its DDE conversations simply by calling *DdeUninitialize*.

Transaction management

Of the five DDEML API functions that specifically control DDE transactions, *DdeClientTransaction* is the key to DDEML transaction management. Except for initiating and terminating conversations, all DDE client transactions originate

with a call to *DdeClientTransaction*, including requests for data, establishing and terminating data links, sending data to a server, and requesting a server to execute a command. In effect, *DdeClientTransaction* subsumes all the functionality implicit in the WM_DDE_REQUEST, WM_DDE_ADVISE, WM_DDE_UNADVISE, WM_DDE_POKE, and WM_DDE_EXECUTE messages in the message-based DDE protocol.

The parameter list to *DdeClientTransaction* includes a transaction identifier whose value (XTYP_REQUEST, XTYP_ADVSTART, XTYP_ADVSTOP, XTYP_POKE, or XTYP_EXECUTE) specifies which transaction you want to perform. Most of the other *DdeClientTransaction* parameters have intuitive uses—to identify a particular conversation, to specify an item name, to point to data or to a command string, to specify a clipboard data format, and to point to a variable that contains a result-code value after the function returns.

The one remaining function parameter, a timeout value, lets you control whether a transaction executes synchronously or asynchronously. If you want a transaction to execute synchronously—that is, to be complete at the time the call to *DdeClientTransaction* returns—you specify a nonzero timeout value in milliseconds. The DDEML will wait for a response from the server for the specified duration of time. If the server fails to respond, *DdeClientTransaction* imposes a timeout and indicates that an error occurred by returning a value of 0. While the DDEML waits for a server response, it enters a message-processing loop that allows application message processing to continue.

The minimum timeout required for successful DDE transaction processing depends on several factors, including the amount of time needed to process data or execute a command and the speed or configuration of the computer system on which Windows is running. A timeout period of 1000 ms (one second) is a reasonable value in most applications. In an application such as a remote communications server, in which response to DDE timeouts can be important, you may want to support user-configurable timeout values. You might also consider designing routines that trap timeout errors and empirically adjust the timeout period toward an optimum value.

The alternative to specifying a timeout period is to use a value of TIMEOUT_ASYNC when you call *DdeClientTransaction*. Doing this causes the DDEML to perform the transaction asynchronously. In this case, the call to *DdeClientTransaction* returns immediately after the transaction is begun, and the DDEML saves the transaction in an internal queue until the server responds to it. For such asynchronous transactions, *DdeClientTransaction* returns a value that uniquely identifies the transaction in the DDEML's internal asynchronous-transaction queue.

When the transaction is completed, the DDEML notifies the client by calling its callback function. One of the parameters passed to the callback function contains the unique transaction-identifier value returned by the original call to *DdeClientTransaction.*

There are two ways to abandon an asynchronous transaction before it is completed. One way is for a client to call *DdeAbandonTransaction.* Another is for the transaction's conversation to be terminated either by the server or by the client. In this case, the DDEML implicitly abandons all pending asynchronous transactions for the terminated conversation.

The fact that the DDEML supports both synchronous and asynchronous transactions implies a mechanism for DDE servers to use the DDEML to block transaction processing transiently. While a server's transaction processing is blocked, the DDEML queues asynchronous transactions until the server is unblocked. A DDE server can block transactions it receives in two ways. One is by calling *DdeEnableCallback* with a command code of EC_DISABLE. If you specify a conversation handle in the call to *DdeEnableCallback*, the DDEML blocks transactions only in the specified conversation. If you specify a null conversation handle, all conversations are affected.

Another method of blocking transactions in a conversation is by returning a special value, CBR_BLOCK, from the server's callback function. Returning CBR_BLOCK places the transaction being processed in the callback function at the front of a queue and causes the DDEML to queue subsequent transactions in the conversation. Regardless of which blocking method you use, the server can unblock and begin to process queued transactions by calling *DdeEnableCallback* with a command code of EC_ENABLEALL (to process all queued transactions) or EC_ENABLEONE (to process only the first queued transaction).

A server cannot block all types of DDEML transactions. A server callback function can identify non-blockable transaction types by testing the XTYPF_NOBLOCK flag, which is part of each transaction-type identifier. A server callback function should not return CBR_BLOCK for any transaction whose type identifier has the XTYPF_NOBLOCK flag set.

DDE applications benefit greatly from the ability to combine asynchronous transaction processing with selective blocking of transactions. In particular, a server that needs time to do a prolonged computation or data transfer can temporarily block DDE transactions from clients until it completes the prolonged operation. For example, a database server might use DDE transaction blocking

to serialize requests for data so that successive database searches do not overlap or collide. A corresponding client application could issue its requests for data asynchronously so that a user's interaction with the application could continue while the server performed a database search.

The most important precaution in the use of asynchronous transactions and transaction blocking is to limit the amount of time that a server blocks incoming transactions. The size of the DDEML's asynchronous-transaction queue is large but not infinite. Prolonged transaction blocking could overflow the queue or potentially overwhelm the server with unnecessarily repeated asynchronous transactions.

The last two API functions that affect DDEML transaction management are *DdePostAdvise* and *DdeSetUserHandle*. A server calls *DdePostAdvise* to transfer updated data items to clients participating in data links. A client calls *DdeSetUserHandle* to associate an application-defined DWORD value with a specific asynchronous transaction. When the transaction is completed and the client callback function is notified, the client can retrieve the associated DWORD value through a call to *DdeQueryConvInfo*.

String management

The DDEML supports a string-management API that resembles Windows' atom-manager API but provides more string-management options. An application calls *DdeCreateStringHandle* to obtain a handle (data type HSZ) that identifies a null-terminated character string. The inverse function, *DdeQueryString*, returns a copy of the string data and the length of the string associated with a particular string handle.

The DDEML associates a usage count with each string handle. *DdeCreateStringHandle* initializes the usage count to 1. You can use two other DDEML API functions, *DdeKeepStringHandle* and *DdeFreeStringHandle*, to increment and decrement the usage count. If a string handle's usage count is decremented to 0, the DDEML releases the memory in which the string is stored and invalidates the string handle.

DDEML-aware programs must use string handles to identify DDE service, topic, and item names. However, you can also choose to use string handles for other purposes because string handles are easier to compare than the strings they represent. For example, a single DDEML function, *DdeCmpStringHandles*, returns an integer that indicates the result of a case-insensitive comparison of the strings the handles represent.

Memory management

The DDEML provides six important API functions that manage shared global memory on behalf of DDE servers and clients. *DdeCreateDataHandle* allocates a block of shared memory and optionally initializes it with data copied from a buffer. *DdeCreateDataHandle* returns a handle (data type HDDEDATA) that is used in the remaining DDEML memory-management functions and in many other DDEML API functions as well.

A program can copy data from a private buffer into a shared memory block by calling *DdeAddData*. The inverse function *DdeGetData* copies data out of a shared memory block into a buffer. A program that needs only to read data from a shared-memory block without copying it can obtain a pointer to the data by calling *DdeAccessData*. Each call to *DdeAccessData* must be paired with a subsequent call to *DdeUnaccessData*, which invalidates the pointer returned by *DdeAccessData*.

Although the DDEML memory-management API resembles Windows' global memory-management API, there is an important difference in regard to freeing allocated memory. In Windows, each call to *GlobalAlloc* should be paired with a call to *GlobalFree*. With the DDEML, the rules are different. If you pass a data handle to a DDEML API function, the DDEML will free the associated memory block at the appropriate time. Similarly, you need not free data handles that the DDEML passes in parameters to an application's callback function.

There are only three situations in which a call to *DdeFreeDataHandle* is necessary. One is when an application allocates a memory block whose handle is never passed to a DDEML API function. Another is when a client receives a data handle as a return value from a call to the *DdeClientTransaction* function. The third is when a memory block is allocated with the HDATA_APPOWNED flag set in the call to *DdeCreateDataHandle*. Such memory blocks are owned by the application that creates them, so they can be reused indefinitely or in multiple DDE conversations. Because the DDEML does not automatically free these blocks of memory, it is the application's responsibility to call the *DdeFreeDataHandle* function for them.

The Callback Function

The callback function in a DDEML application resembles a window function. Like a window function, a DDEML callback function is an exported PASCAL FAR function in which the first parameter identifies the kind of action the function is expected to carry out. In the following example, the EXPENTRY declaration defines *DdeCallback* as an exported PASCAL FAR function.

```
HDDEDATA EXPENTRY
DdeCallback( WORD wType, WORD wFmt, HCONV hConv,
             HSZ hsz1, HSZ hsz2, HDDEDATA hDDEData,
             DWORD dwData1, DWORD dwData2 )
```

The values represented in the callback function parameters vary according to the transaction type, as shown in Figure 6-11.

Transaction-Type Identifier	***Received By***	***Parameter***	***Return Value***
XTYP_ADVDATA Updates a data item in a data link	Client	*wFmt*: clipboard data-format *hConv*: conversation handle *hsz1*: topic name *hsz2*: item name *hDDEData*: handle to data from server	Transaction result flag
XTYP_ADVREQ Requests updated data item in a data link	Server	*wFmt*: clipboard data-format *hConv*: conversation handle *hsz1*: topic name *hsz2*: item name	Handle to updated data item
XTYP_ADVSTART Initiates a data link ("advise")	Server	*wFmt*: clipboard data-format *hConv*: conversation handle *hsz1*: topic name *hsz2*: item name	TRUE to start data link for the specified item
XTYP_ADVSTOP Terminates a data link ("unadvise")	Server	*wFmt*: clipboard data-format *hConv*: conversation handle *hsz1*: topic name *hsz2*: item name	0
XTYP_EXECUTE Requests command execution	Server	*hConv*: conversation handle *hsz1*: topic name *hDDEData*: command string	Transaction result flag

Figure 6-11. *(continued)*
Transaction-type identifiers and parameters for DDEML callback functions.

Figure 6-11. *continued*

Transaction-Type Identifier	***Received By***	***Parameter***	***Return Value***
XTYP_CONNECT Initiates conversation	Server	*hsz1*: topic name *hsz2*: service name *dwData1*: conversation context (PCONVCONTEXT)	TRUE to support a conversation on the specified topic
XTYP_CONNECT-_CONFIRM Confirms initiated conversation	Server	*hConv*: conversation handle *hsz1*: topic name *hsz2*: server name *dwData2*: TRUE if same server and client instance	0
XTYP_XACT_COMPLETE Completes asynchronous transaction	Client	*wFmt*: clipboard data-format *hConv*: conversation handle *hsz1*: topic name *hsz2*: item name *hDDEData*: handle to data from server *dwData1*: transaction ID	0
XTYP_POKE Transfers data item from client to server	Server	*wFmt*: clipboard data format *hConv*: conversation handle *hsz1*: topic name *hsz2*: item name *hDDEData*: handle to data from client	Transaction result flag
XTYP_REGISTER Indicates newly-registered service name	Server, Client	*hsz1*: service name *hsz2*: unique server-instance name	0

(continued)

Figure 6-11. *continued*

Transaction-Type Identifier	***Received By***	***Parameter***	***Return Value***
XTYP_REQUEST Requests a data item	Server	*wFmt*: clipboard data format *hConv*: conversation handle *hsz1*: topic name *hsz2*: item name	Handle to requested data
XTYP_DISCONNECT Confirms termination of a conversation	Server, Client	*hConv*: conversation handle	0
XTYP_UNREGISTER Indicates unregistration of a service name	Server, Client	*hsz1*: service name *hsz2*: unique service-instance name	0
XTYP_WILDCONNECT Requests initiation of new conversation(s) using wildcard specification	Server	*hsz1*: topic name *or* wildcard *hsz2*: service name *or* wildcard *dwData1*: conversation context (PCONVCONTEXT)	Handle to list of matching service-topic names
XTYP_MONITOR Used only in DDEML monitor applications; see the DDEML documentation for details			

You can design a DDEML callback function with a C-language *switch* statement whose cases correspond to the possible values of the transaction-type identifier. However, the structure of the DDEML transaction-type identifier lets you use other flow-of-control structures besides the *switch* statement in a DDEML callback function.

The transaction-type identifier is actually composed of an index value combined with one of the flags shown in Figure 6-12. These flags classify each transaction according to the meaning of the value that the callback function is expected to return.

Flag	***Callback Function Returns***
XCLASS_BOOL	TRUE or FALSE
XCLASS_DATA	Data handle (HDDEDATA)
XCLASS_FLAGS	Transaction result flag (DDE_FACK, DDE_FBUSY, or DDE_NOTPROCESSED)
XCLASS_NOTIFICATION	0

Figure 6-12.
Transaction-class flags used in transaction-type identifiers for a DDEML callback function. The function returns a value of data-type HDDEDATA regardless of the meaning of the return value.

There is an additional flag, XTYPF_NOBLOCK, that indicates whether a callback function can return CBR_BLOCK to block a transaction. For example, this is how the transaction-type identifier XTYP_CONNECT is defined in DDEML.H:

```
#define XTYP_CONNECT    (0x0060 | XCLASS_BOOL | XTYPF_NOBLOCK)
```

In this definition, the value 0x0060 is an index value that uniquely identifies the transaction type.

You can therefore structure a DDEML callback function according to the transaction class:

```
switch( wType & XCLASS_MASK )
{
 case XCLASS_BOOL:
  :
  break;

 case XCLASS_DATA:
  :
  break;

 case XCLASS_FLAGS:
  :
  break;

 case XCLASS_NOTIFICATION:
  :
  break;
}
```

You can also take advantage of the fact that the index values that are used in the transaction-type definitions follow a numerical sequence from 0x0000 through 0x00F0. This allows you to use a jump table instead of a *switch* statement to select among transaction-processing functions. The sample callback function in Figure 6-13 illustrates this technique.

```
/* The following typedefs address the arguments to a DDEML
   callback function. The data structure maps the
   arguments as they appear on the stack according to the
   PASCAL parameter-passing convention.*/

typedef struct
{
    DWORD      dwData2;
    DWORD      dwData1;
    HDDEDATA   hDDEData;
    HSZ        hsz2;
    HSZ        hsz1;
    HCONV      hConv;
    WORD       wFmt;
    WORD       wType;
}
    XACTPARAMS;

typedef XACTPARAMS *    PXACTPARAMS;

typedef HDDEDATA    (*PFNXACT)( PXACTPARAMS );


/* prototypes for transaction-processing functions */
static HDDEDATA     XactAdvdata( PXACTPARAMS );
static HDDEDATA     XactXactComplete( PXACTPARAMS );
static HDDEDATA     XactDisconnect( PXACTPARAMS );
static HDDEDATA     XactIgnore( PXACTPARAMS );
static HDDEDATA     XactError( PXACTPARAMS );

/* jump table */
static PFNXACT pfnCallback[] =
        {
          XactError,          /* 00: XTYP_ERROR */
          XactAdvdata,        /* 01: XTYP_ADVDATA */
          XactIgnore,         /* 02: XTYP_ADVREQ */
          XactIgnore,         /* 03: XTYP_ADVSTART */
          XactIgnore,         /* 04: XTYP_ADVSTOP */
          XactIgnore,         /* 05: XTYP_EXECUTE */
          XactIgnore,         /* 06: XTYP_CONNECT */
          XactIgnore,         /* 07: XTYP_CONNECT_CONFIRM */
```

Figure 6-13. *(continued)*
Using a jump table in a DDEML callback function.

Figure 6-13. *continued*

```
            XactXactComplete,     /* 08: XTYP_XACT_COMPLETE */
            XactIgnore,           /* 09: XTYP_POKE */
            XactIgnore,           /* 0A: XTYP_REGISTER */
            XactIgnore,           /* 0B: XTYP_REQUEST */
            XactDisconnect,       /* 0C: XTYP_DISCONNECT */
            XactIgnore,           /* 0D: XTYP_UNREGISTER  */
            XactIgnore,           /* 0E: XTYP_WILDCONNECT */
            XactIgnore            /* 0F: XTYP_MONITOR */
        };


/*************************************************************
 *                                                           *
 * DdeCallback                                               *
 *                                                           *
 *************************************************************/

HDDEDATA EXPENTRY
DdeCallback( WORD wType, WORD wFmt, HCONV hConv,
               HSZ hsz1, HSZ hsz2, HDDEDATA hDDEData,
               DWORD dwData1, DWORD dwData2 )
{
    int   nIndex;


    /* extract index value from transaction-type ID */
    nIndex = (wType & XTYP_MASK) >> XTYP_SHIFT;

    /* jump through the table of function pointers */
    return (*pfnCallback[nIndex])( (PXACTPARAMS)&dwData2 );
}
```

Of the 15 transaction types used in DDEML callback functions, one (XTYP_MONITOR) is used only in DDEML monitor applications such as DDESPY. Of the remaining 14, nine are reserved for servers only and two for clients only. The other three (XTYP_REGISTER, XTYP_DISCONNECT, and XTYP_UNREGISTER) are used in both server and client callback functions. You can design a callback function to process server transactions, client transactions, or both. In any case, remember to specify the appropriate command flags (APPCLASS_STANDARD with or without APPCMD_CLIENTONLY) when you call *DdeInitialize* to pass the callback function's address to the DDEML.

XTYP_REGISTER and XTYP_UNREGISTER

The DDEML uses these two transaction types to inform an application that another application has registered or unregistered a DDE service name. The purpose of these transactions is to alert DDE applications to the appearance or disappearance of services supported by DDE servers.

XTYP_CONNECT, XTYP_WILDCONNECT, XTYP_CONNECT_CONFIRM, and XTYP_DISCONNECT

A DDE server application's callback function receives XTYP_CONNECT when a client attempts to initiate a conversation on a specified service and topic. The callback function should return TRUE (a nonzero value) if the server supports the specified service and topic.

Similarly, a server receives XTYP_WILDCONNECT when a client wants to initiate conversations using a wildcard service or a topic specification. In this case, the server callback function must fill a block of shared global memory with an array of string-handle pairs (data type HSZPAIR) that enumerates service-topic combinations that match the wildcard specification. A null HSZPAIR indicates the end of the list. The return value from the callback function is the data handle (data type HDDEDATA) to the memory block containing the HSZPAIR list.

Each time the DDEML successfully establishes a conversation, it calls the server callback function with a transaction type of XTYP_CONNECT_CONFIRM. With this transaction type, the DDEML passes a conversation handle (HCONV) that identifies the conversation in subsequent transactions, along with string handles that indicate the service and topic names for which the conversation was established. Although a server may save these handles for future reference, it is not strictly necessary to do so. The other callback-function transactions all provide these handles as function parameters whenever they are needed.

The DDEML uses the XTYP_DISCONNECT transaction type to notify both server and client applications that a conversation has terminated. This transaction type exists only for the convenience of the DDE application. A callback function should return a value of 0 in response to this transaction.

Although the DDEML registers service names and passes string handles to service, topic, and item names as callback-function parameters, each DDE server and client application is responsible for maintaining its own lists of the service, topic, and item names that it supports. Use linked data structures to keep track of names so that you can easily associate a list of topic names with each service name supported by a server application. You can use the same technique to associate a list of item names and data pointers with each topic name.

XTYP_REQUEST and XTYP_ADVREQ

A server's callback function receives these two transaction types whenever the DDEML wants the server to transfer a data item to a client. A server receives XTYP_REQUEST when a client calls *DdeClientTransaction* to request data. The XTYP_ADVREQ transaction type is used when a server application calls *DdePostAdvise* to transmit an updated data item to a client in a data link. In both cases, the parameters passed to the callback function include the conversation handle, string handles to the topic and item names, and the clipboard data format for the requested data. The server callback function should return a data handle (HDDEDATA) to a shared memory block that contains the requested data.

XTYP_POKE

This transaction type notifies a server that an unsolicited data item has been transmitted from a client. The callback function parameters include a data handle for the shared-memory block that contains the data, the conversation handle, string handles to the topic and item names, and the clipboard data format. The server's callback function must return a flag (DDE_FACK, DDE_FBUSY, or DDE_NOTPROCESSED) that indicates whether the server accepted the data item.

XTYP_EXECUTE

The XTYP_EXECUTE transaction type is similar to XTYP_POKE. The difference is that the data handle passed to the server's callback function refers to a memory block that contains a null-terminated command string. The flag returned by the callback function (again, DDE_FACK, DDE_FBUSY, or DDE_NOTPROCESSED) indicates whether the server accepted the command.

XTYP_ADVSTART and XTYP_ADVSTOP

The DDEML calls a server callback function with XTYP_ADVSTART when a client attempts to start a data link with the server. The function parameters associated with XTYP_ADVSTART include a conversation handle, string handles for the topic and item names for which the data link is requested, and a clipboard data format. The callback function must return TRUE if the server will support the requested data link.

The XTYP_ADVSTOP transaction type notifies a server that a data link has been terminated by a client. The server application need not do anything in response. The callback function must return 0.

XTYP_ADVDATA and XTYP_XACT_COMPLETE

These two transaction types are the only ones processed exclusively by client-application callback functions. The DDEML uses XTYP_ADVDATA to pass a data item from a server to a client in a data link. The XTYP_ADVDATA transaction type is just like XTYP_POKE. The callback-function parameters specify a conversation handle, topic and item names, a clipboard data format, and a data handle. The client callback function must return DDE_FACK, DDE_FBUSY, or DDE_NOTPROCESSED to indicate whether it accepted the data.

The DDEML uses the XTYP_XACT_COMPLETE transaction type to notify a client application that an asynchronous transaction has completed. When a client specifies TIMEOUT_ASYNC instead of a timeout period in a call to *DdeClientTransaction*, the DDEML queues the transaction until the server is free to process it. The server's response is always returned to the client through XTYP_XACT_COMPLETE, regardless of the type of transaction carried out.

The function parameters associated with XTYP_XACT_COMPLETE include the usual conversation handle, string handles, and clipboard data type. The crucial parameter, however, is the first DWORD parameter, which contains the unique transaction identifier that was returned previously as a result value by *DdeClientTransaction.*

The transaction identifer is important because the client can use it in a call to *DdeQueryConvInfo*, which fills a CONVINFO data structure with data that describe the asynchronous transaction whose completion was signaled by XTYP_XACT_COMPLETE. Armed with the information in the CONVINFO data structure, the client callback function can carry out the same response to the completed asynchronous transaction as it might have if the transaction had been processed synchronously, as shown in Figure 6-14.

Initiating a Conversation

All of the DDEML callback-function transaction types make sense when you consider how they are used in relation to the DDEML API functions. For example, to initiate a conversation, a DDEML client calls *DdeConnect* or *DdeConnectList.* For each potential conversation—that is, for each service-topic pair requested by the client—the DDEML calls the appropriate server callback function with an XTYP_CONNECT or XTYP_WILDCONNECT transaction type.

```
/* (called when a client receives XTYP_XACT_COMPLETE) */
HDDEDATA XactComplete( HCONV hconv, DWORD dwData1 )
{

    CONVINFO    ci;


    /* Use the transaction ID to obtain information */
    ci.cb = sizeof(CONVINFO);
    DdeQueryConvInfo( hConv, dwData1, &ci );

    /* Respond to completion of the asynchronous transaction */
    switch( ci.wType )
    {
      case XTYP_ADVSTART:
      case XTYP_ADVSTART | XTYPF_NODATA:
      case XTYP_ADVSTART | XTYPF_ACKREQ:
      case XTYP_ADVSTART | XTYPF_NODATA | XTYPF_ACKREQ:
          :
        break;

      case XTYP_ADVSTOP:
          :
        break;

      case XTYP_EXECUTE:
          :
        break;

      case XTYP_REQUEST:
          :
        break;

      default:
        break;
    }

    /* if an error occurred, display it */
    if( ci.LastError )
      DisplayErrorMessage( ... );

    return 0;
}
```

Figure 6-14.

A client callback function calls DdeQueryConvInfo *in order to process XTYP_XACT_COMPLETE. The transaction identifier is contained in the first DWORD parameter passed to the callback function (*dwData1*). The original transaction type of the completed asynchronous transaction is found in* wType *in the CONVINFO data structure.*

For each successfully initiated conversation, the DDEML passes a conversation handle to both client and server. The DDEML does this for the client through the return value from *DdeConnect* or *DdeConnectList.* On the server side, the DDEML calls the server's callback function with an XTYP_CONNECT_CONFIRM transaction type.

Requesting Data from a Server

To request data from a server, a client application calls *DdeClientTransaction* with a transaction type of XTYP_REQUEST. The DDEML calls the corresponding server's callback function with XTYP_REQUEST. The data handle returned by the server gets back to the client in one of two ways. If the transaction is performed synchronously, the data handle appears as the return value from *DdeClientTransaction.* For an asynchronous transaction, the DDEML calls the client's callback function with XTYP_XACT_COMPLETE, with the data handle as one of the callback-function parameters.

Establishing a Data Link

A client application establishes a data link by calling *DdeClientTransaction* with the XTYP_ADVSTART transaction type. The DDEML calls the server's callback function with the same transaction type. If the transaction is synchronous, the server's response is reflected in *DdeClientTransaction*'s return value. If the transaction is asynchronous, the client receives XTYP_XACT_COMPLETE from the DDEML. The client then checks *wLastError* in the CONVINFO data structure to determine whether the data link was established.

Sending Data to a Server

A client sends a data item to a server by calling *DdeClientTransaction* and specifying the XTYP_POKE transaction type. The server's callback function receives the data in an XTYP_POKE transaction. The DDEML returns the server's response to the client either through the return value from *DdeClientTransaction* or through XTYP_XACT_COMPLETE and the *wLastError* value in the CONVINFO data structure.

Executing a Command

To execute a command, a client application calls *DdeClientTransaction* with the XTYP_EXECUTE transaction type. The server receives XTYP_EXECUTE and

returns DDE_FACK, DDE_FBUSY, or DDE_NOTPROCESSED, which the client receives either as a result value from *DdeClientTransaction* for a synchronous transaction or via XTYP_XACT_COMPLETE for an asynchronous transaction.

Design Issues in DDE Applications

The specifications for message-based DDE and for the DDEML API explicitly describe the low-level transaction processing required in a DDE conversation. Nevertheless, there is more to designing a successful DDE application than simply supporting a full set of DDE transactions. For example, two DDE applications must agree on the meanings of service and topic names, on what data formats to support, and on meaningful ways to specify and obtain results from executable commands.

Selecting Service, Topic, and Item Names

The DDE specification provides no guidance in choosing service, topic, and item names. Consequently, different applications use service, topic, and item names to represent different things. For example, Microsoft Excel uses an application name (*Excel*) as a service name, a spreadsheet name (*Sheet1*) as a topic, and a spreadsheet cell identifier (*R1C1*) as an item. However, an application that uses DDE to provide access to a relational database might use the service name to identify a database, the topic name to identify a database table, and item names to designate columns in the table.

Two applications can establish a DDE conversation only if they agree on the meanings of service, topic, and item names. You should therefore accompany any DDE application you design with clear documentation of how the application uses service, topic, and item names. You can also automate the process to some extent by including support for the System topic in any DDE server application you design. Yet another approach is to design your DDE applications to use the clipboard to transfer service, topic, and item names to other DDE applications, using the paste link method described in the DDE specification.

In a DDE paste link, a server places a specified service, topic, and item name on the clipboard. A DDE client copies the names from the clipboard and uses them to initiate a conversation on the specified topic and a data link for the specified item. The server should format the service, topic, and item names as a sequence of three null-terminated strings, followed by an additional null byte:

```
Service\0Topic\0Item\0\0
```

Both server and client applications should register the "Link" clipboard format and use it to access the service, topic, and item names on the clipboard. The server side of the paste link should be associated with a Copy command in an Edit drop-down menu. A client should use a Paste Link command in an Edit menu.

Supporting the System Topic

The Windows SDK documentation recommends that all DDE servers support a consistent set of data items under the System topic, as shown in Figure 6-15. These data items describe the topics supported by a DDE server. They can also provide a general indication of the server's status. All the System data items use the CF_TEXT format, which makes it a bit easier for an application to describe a DDE server's configuration or status to a human user.

Item Name	*Description*
SysItems	A list of items supported by the DDE server application under the System topic; for example `"SysItems\tTopics\tStatus\tFormats"`
Topics	A list of topics supported by the server
Status	Description of the current status of the server application; for example `"Ready"` and `"Busy"`
Formats	A list of clipboard data formats potentially supported by the server; for example `"TEXT\tCSV\tLink"`
TopicItemList	A list of items used by all topics except the System topic
Help	A string containing plain-text information that would help a user access the server application

Figure 6-15.
Data items supported under the System topic. All data items have the CF_TEXT format (null-terminated strings that use tabs as separators).

The Formats data item can be particularly useful to DDE client applications. A client application can parse the list of clipboard formats in this item to determine which clipboard format to specify in subsequent requests for data from a DDE server. There is one complication, however. When you design a DDE client to use the Formats item, the client must distinguish between the predefined clipboard formats, which have symbolic values defined in WINDOWS.H, and registered clipboard formats, which have values obtained from a call to *RegisterClipboardFormat.*

By convention, a server represents the predefined clipboard formats with strings that correspond to the CF_ symbols defined in WINDOWS.H, but with the "CF_" prefix removed. (For example, a server indicates that it supports the CF_TEXT format by including "TEXT" in the Formats data item.) When a client parses the Formats item, it must test whether each format name is one of the predefined clipboard-format names. If it is, the client should use the corresponding CF_ value in WINDOWS.H; if not, the client should call *RegisterClipboardFormat* to obtain a clipboard-format value.

Executing Commands

The DDE specification defines a simple syntax for command strings used in a DDEML XTYP_EXECUTE transaction or in a WM_DDE_EXECUTE message. In this syntax, pairs of square brackets delimit individual commands. Commands themselves consist of alphanumeric verbs with optional comma-separated parameter lists enclosed in parentheses. For example, the following DDE command sequence requests Microsoft Excel to display a message box and then beep.

```
[alert("Message from DDE client",2)] [beep(0)]
```

These two commands could be sent from a DDE client to Excel either in a single DDE transaction or in a sequence of two separate transactions.

The DDE specification is unclear about whether a command string can be associated with a particular topic. Traditionally, only the System topic responds to commands, but you may find it useful to execute commands in the context of conversations on topics other than the System. This allows applications to obtain different results by executing the same command for different topics. For example, imagine a database application in which a topic corresponds to a table in a relational database. A "select" command for this topic could implicitly refer to the particular database table represented in the topic; the same command for a different topic—that is, for a different database table—would produce a different result set.

There is an additional design problem in acknowledging WM_DDE_EXECUTE messages (with message-based DDE) or XTYP_EXECUTE transactions (with the DDEML). The problem arises because commands may take a long time to execute. For example, a command might result in a server displaying a message and waiting for user input. In this situation, you must carefully consider how the server and client cooperate in acknowledging the command.

If the server acknowledges receipt of the command when it receives it, the client can infer only that the command was successfully received. The client cannot assume that the command has actually been executed because the client might receive the server's acknowledgement long before the server actually finishes executing the command. On the other hand, if the server defers acknowledgment until after it processes the command, the client might wait needlessly for a slow server to execute a command.

One solution to this dilemma is for the server to support a special "Command Status" data item that contains the status of any command the server is executing. Before posting any commands to the server, the client establishes a data link with the server for the Command Status item. Then, when the client posts a command to the server, the server can acknowledge receipt of the command directly. When the server finishes executing the command, it can update the Command Status item in the data link. The client can thus determine when a command has been successfully received and when it has actually been executed.

Beyond DDE

The DDE protocol, whether embodied in Windows messages or in the DDEML, is essentially a handshaking protocol for inter-application transfers of data. DDE makes it possible for applications to exchange multiple data items in an orderly way. DDE's service/topic/item naming hierarchy is adaptable to a variety of data-exchange scenarios, including network communications and database applications.

Nevertheless, DDE is not a good tool for binding different applications together interactively. Doing so requires you to know low-level details about the applications you use, including service, topic, and item names or registered clipboard formats. You also must know how to access DDE through the applications you are using. For example, if you are using Microsoft Excel, you must understand Excel's macro language as well as DDE. A higher-level approach to interprocess communication would be better than DDE for many Windows users.

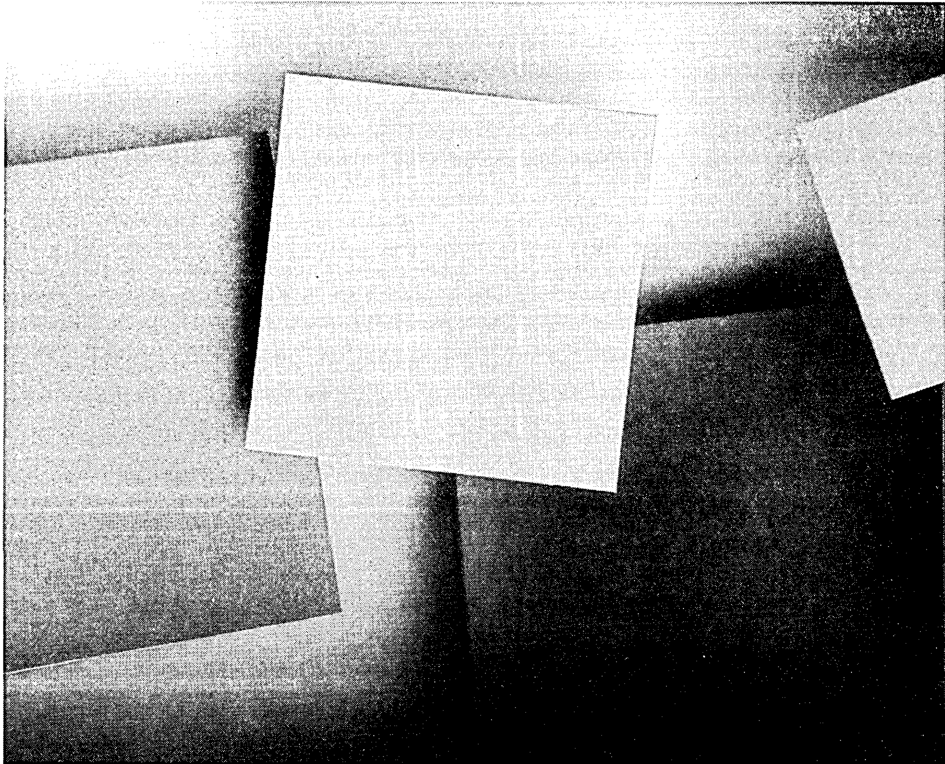
Microsoft's approach to supporting high-level interprocess communication is a protocol called Object Linking and Embedding (OLE). Unlike DDE, which is designed to support data transfer between applications, OLE supports functional links between documents such as spreadsheets, charts, or word processing documents. You can think of an OLE document as a compound document that can contain any number of different data objects, each of which is associated with an application that can be used to edit it. From a Windows user's point of view, such functionality is much more intuitive than that supported in DDE.

The "linking" part of OLE refers to dynamic links between data objects in an OLE document and the applications that manipulate the data objects. For example, a Windows user can use the mouse to click on a linked data object in a compound document and thereby execute the application with which the data can be edited. In contrast, "embedding" refers to storing a data object in a compound document without maintaining a dynamic link to another application.

From a programmer's perspective, a choice between OLE and DDE depends on what functionality an application is designed to support. For low-level data transfers that do not require direct intervention, DDE is an appropriate choice. For high-level, user-controlled data links between Windows applications, you should consider using OLE.

7

Problems and Solutions



This chapter is a collection of Windows programming techniques. All are "advanced" in that they make more sense when you are comfortable programming in the Windows environment, but none are tricks or secrets. Everything is based on an understanding of how things work in Windows.

Control Variations

The built-in control classes—particularly the list-box, edit, and combo-box classes—provide a great deal of ready-to-use functionality. Many good Windows applications rely entirely on the predefined control classes.

Nevertheless, the predefined control classes don't do everything. If you want a control that looks like one of the built-in control classes but that behaves somewhat differently, you must decide whether to build a look-alike variation from scratch as a custom control or to work with a built-in control and add the functionality you need.

Controls with Thick Frames

Imagine that you want to design a list-box control with a title bar and fat borders that let the user move and size the control. You might suppose that you could use the WS_CAPTION, WS_SYSMENU, and WS_THICKFRAME window styles to create such a window. For example, the list box shown in Figure 7-1 on the following page can be created by using the following *CreateWindow* call:

```
CreateWindow( "ListBox",
              "Popup ListBox",
              WS_CAPTION | WS_POPUP | WS_THICKFRAME |
                WS_SYSMENU | WS_VSCROLL | WS_VISIBLE,
              200, 200, 200, 200,
              hWnd,
              0,
              hInstance,
              NULL );
```

The problem with this technique is that all the predefined control classes do not work properly with all possible window styles in all versions of Windows. The predefined controls were not designed to be resized or to serve as pop-up windows. You may find that windows such as the pop-up list box shown in Figure 7-1 do not flash when they receive the focus or fail to draw their non-client areas properly. Figure 7-1 is a case in point—the list-box class window function in Windows 3.0 has ignored the WS_THICKFRAME and WS_CAPTION styles and made the client area too small, incorrectly clipping the last item in the list box.

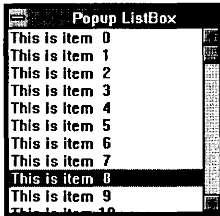


Figure 7-1.
A list-box control with the WS_THICKFRAME and WS_CAPTION styles.

A more reliable way to support a control with a caption bar or a thick frame is to frame the control in a parent window that has the desired window style. The parent can process WM_SIZE messages to size the child-window control appropriately within the parent's client area. All predefined control classes respond properly to WM_SIZE messages, so the parent can call the *MoveWindow* function, which sends a WM_SIZE message, to control the size of a child-window control.

Figure 7-2 shows how to use this method. The top-level window in the application is the parent of an edit control. The top-level window function, *TopLevelWndFn*, processes the WM_SIZE message by calling *MoveWindow* to resize the edit control. The parent-child combination has the visual appearance of a single edit-class window that can be moved and sized.

```
#***************************************************************************
#                                                                          *
# NMAKE description for OVEDIT.EXE                                         *
#                                                                          *
#***************************************************************************

.c.obj:
    cl /AM /c /G2sw /Osw /W4 /Zlp $*.c

ALL:            ovedit.exe

ovedit.obj:     ovedit.c

ovedit.exe:     ovedit.obj ovedit.def
    link /al:16 /nod /noe ovedit, , , libw mlibcew, ovedit.def
    rc ovedit.exe
```

Figure 7-2. *(continued)*
Source code for OVEDIT.EXE.

Figure 7-2. *continued*

```
/**************************************************************************
 *                                                                        *
 * OVEDIT.C                                                               *
 *                                                                        *
 * Exports:      TopLevelWndFn                                            *
 *                                                                        *
 **************************************************************************/

#define     NOCOMM
#include    <windows.h>

#define IDEDIT  100


/*** FUNCTION PROTOTYPES ***/

LONG PASCAL FAR TopLevelWndFn( HWND, WORD, WORD, LONG );

static HWND     Init( HANDLE, HANDLE, int );


/*** GLOBAL VARIABLES ***/

char        szTopLevelClass[] = "OvEdit:TopLevel";


/**************************************************************************
 *                                                                        *
 * WinMain                                                                *
 *                                                                        *
 **************************************************************************/

int PASCAL
WinMain( HANDLE hInst, HANDLE hPrevInst, LPSTR lpszCmdLine, int nCmdShow )
{
    HWND    hWnd;
    MSG     msg;


    hWnd = Init( hInst, hPrevInst, nCmdShow );
    if( !hWnd )
      return 0;
```

(continued)

Figure 7-2. *continued*

```
    while( GetMessage( &msg, 0, 0, 0 ) )
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }

    return msg.wParam;
}


/*************************************************************************
*                                                                       *
* Init                                                                  *
*                                                                       *
*************************************************************************/

static HWND Init( HANDLE hInst, HANDLE hPrevInst, int nCmdShow )
{
    WNDCLASS    wc;
    HWND        hWnd;


    if( 0 == hPrevInst )
    }
      /* register the top-level window class */
      wc.lpszClassName   = szTopLevelClass;
      wc.hInstance       = hInst;
      wc.lpfnWndProc     = TopLevelWndFn;
      wc.hCursor         = LoadCursor( 0, IDC_ARROW );
      wc.hIcon           = LoadIcon( 0, IDI_APPLICATION );
      wc.lpszMenuName    = NULL;
      wc.hbrBackground   = COLOR_WINDOW+1;
      wc.style           = CS_HREDRAW | CS_VREDRAW;
      wc.cbClsExtra      = 0;
      wc.cbWndExtra      = 0;

      if( !Register Class( &wc ) )
        return 0;                        /* return 0 if unsuccessful */
    }

    /* create the top-level window */
    hWnd = CreateWindow( szTopLevelClass,
                         "OvEdit",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                         0,
```

(continued)

Figure 7-2. *continued*

```
                        0,
                        hInst,
                        NULL );

    /* create the edit control */
    CreateWindow( "Edit",
                        "",
                        WS_CHILD | WS_VSCROLL | WS_HSCROLL | WS_VISIBLE |
                          ES_MULTILINE,
                        0, 0, 0, 0,
                        hWnd,
                        IDEDIT,
                        hInst,
                        NULL );

    /* display the top-level window */
    ShowWindow( hWnd, nCmdShow );
    UpdateWindow( hWnd );

    return hWnd;
}


/************************************************************************
*                                                                       *
* TopLevelWndFn                                                         *
*                                                                       *
************************************************************************/

LONG PASCAL FAR
TopLevelWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bDWP = FALSE;


    switch( wMsg )
    {
      case WM_SIZE:
        MoveWindow( GetDlgItem( hWnd, IDEDIT ),
                        0, 0,
                        LOWORD(lParam), HIWORD(lParam),
                        FALSE );
        break;
```

(continued)

Figure 7-2. *continued*

```
        case WM_SETFOCUS:
          SetFocus( GetDlgItem( hWnd, IDEDIT ) );
          break;

        case WM_DESTROY:
          PostQuitMessage( 0 );
          break;

        default:
          bDWP = TRUE;
          break;
      }

    if( bDWP )
      lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

    return lRVal;
}
```

```
;*************************************************************************
;                                                                        *
; OVEDIT.DEF module-definition file                                      *
;                                                                        *
;*************************************************************************

NAME            OVEDIT
DESCRIPTION     'OVEDIT.EXE version 1.0'
EXETYPE         WINDOWS

STUB            'WINSTUB.EXE'

CODE            MOVEABLE LOADONCALL DISCARDABLE
DATA            MOVEABLE MULTIPLE PRELOAD

SEGMENTS        _TEXT   MOVEABLE PRELOAD DISCARDABLE

HEAPSIZE        1024
STACKSIZE       5120

EXPORTS         TopLevelWndFn
```

You need not force the size of the child-window control to equal the size of the parent's client area. You might instead hide the child-window control whenever the size of the parent's client area reaches a predefined threshold. You could also keep the control's size proportional to the size of the parent. Windows' Notepad application illustrates this technique, as shown in Figure 7-3. In Notepad, the size of the edit control is somewhat smaller than the size of the parent's client area. In this way, Notepad maintains a visible margin between edited text and the edge of the parent window. As you change the size of the parent, Notepad resizes the edit control to maintain the margin.

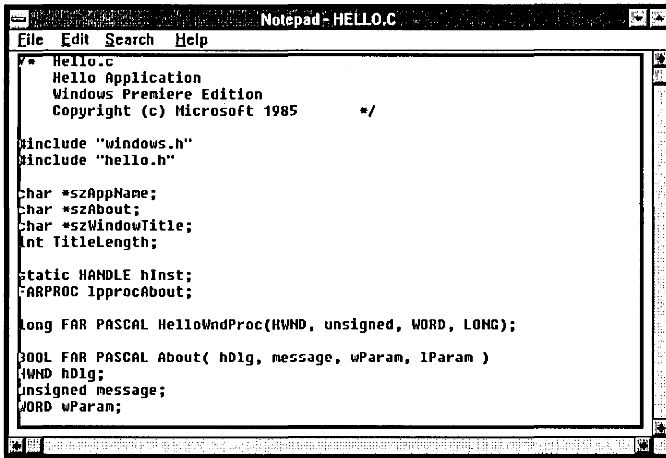


Figure 7-3.
In Notepad, the edit control (outlined in black) is always somewhat smaller than its parent window's client area.

Filtering Messages

Another way to create variations of the predefined control classes is to use message filtering or subclassing. The only problem with this approach is that Microsoft does not document how specific messages are processed by each of Windows' predefined control classes. This means that a message-filtering technique that works in the current version of Windows might fail in a future version should Microsoft change the way predefined control classes process messages. Nevertheless, the technique is powerful and well worth investigating if your application requires only a simple variant of one of the predefined controls.

For example, you can design read-only versions of the default edit, list-box, and combo-box controls by trapping the keyboard and mouse messages that are normally processed by the controls' window functions. By blocking normal keyboard and mouse input, you create controls whose contents cannot be changed interactively.

The source code for ROEDIT.DLL, in Figure 7-4, defines a read-only variation of the default edit control class. The read-only class is named *ROEdit.* You can use *ROEdit* controls just as you would use edit controls. The only difference is that you must be certain to load ROEDIT.DLL with a call to *LoadLibrary* before you create any *ROEdit* controls, as shown in Figure 7-5 on page 220.

```
#************************************************************************
#                                                                       *
# NMAKE description for ROEDIT.DLL                                      *
#                                                                       *
#************************************************************************

.c.obj:
    cl /AMw /c /D _WINDOWS /D _WINDLL /G2sw /Osw /W4 /Zlp $*.c

ALL:                roedit.dll

init.obj:           init.c roedit.h

roedit.obj:         roedit.c roedit.h

wep.obj:            wep.c


roedit.dll:         init.obj roedit.obj wep.obj roedit.def
      link /al:16 /nod /noe libentry init roedit wep, roedit.dll, , \
        libw mdllcew, roedit.def
      rc roedit.dll
```

Figure 7-4. *(continued)*
Source code for ROEDIT.DLL.

Figure 7-4. *continued*

```
/*************************************************************************
*                                                                        *
* INIT.C                                                                 *
*                                                                        *
*************************************************************************/

#define NOCOMM
#include    <windows.h>
#include    "roedit.h"


/*** GLOBAL VARIABLES ***/

HANDLE          hDLLInst;
FARPROC         pDefEditWndFn;
static char     szROEditClass[] = "ROEdit";


/*** FUNCTION PROTOTYPES ***/

BOOL PASCAL FAR     LibMain( HANDLE, WORD, WORD, LPSTR );

static BOOL         RegisterEditSubclass( HANDLE );


/*************************************************************************
*                                                                        *
* LibMain                                                                *
*                                                                        *
*************************************************************************/

BOOL PASCAL FAR
LibMain( HANDLE hInst, WORD wDS, WORD wHeapSize, LPSTR lpCmdTail )
{
    BOOL    bRVal;

    /* if LibEntry has called LocalInit, unlock the default data segment */
    if( wHeapSize )
      UnlockSegment( wDS );

    /* save the DLL instance handle in a global variable */
    hDLLInst = hInst;
```

(continued)

Figure 7-4. *continued*

```
    /* register window classes */
    bRVal = RegisterEditSubclass( hInst );

    return bRVal;
}


/************************************************************************
*                                                                       *
* RegisterEditSubclass                                                  *
*   Returns TRUE if the filter function is successfully registered.     *
*                                                                       *
************************************************************************/

static BOOL RegisterEditSubclass( HANDLE hLibInst )
{
    BOOL        bRVal = FALSE;
    WNDCLASS    wc;


    if( hLibInst )
    {
      /* get default WNDCLASS values for the edit class */
      GetClassInfo( 0, "Edit", &wc );

      /* save the address of the edit class window function */
      pDefEditWndFn = (FARPROC)wc.lpfnWndProc;

      wc.hInstance       = hLibInst;
      wc.lpszClassName   = szROEditClass;
      wc.lpfnWndProc     = ROEditWndFn;
      wc.style           != CS_GLOBALCLASS;

      bRVal = Register Class( &wc );
    }

    return bRVal;
}
```

(continued)

Figure 7-4. *continued*

```
/****************************************************************************
 *                                                                          *
 * ROEDIT.C                                                                 *
 *                                                                          *
 * Exports:  ROEditWndFn                                                    *
 *                                                                          *
 ****************************************************************************/

#define NOCOMM
#include    <windows.h>
#include    "roedit.h"


/*** FUNCTION PROTOTYPES ***/

static void     MsgKeyDown( HWND, WORD );


/*** GLOBAL VARIABLES ***/

extern FARPROC  pDefEditWndFn;          /* (defined in INIT.C) */


/****************************************************************************
 *                                                                          *
 * ROEditWndFn                                                              *
 *                                                                          *
 ****************************************************************************/

LONG PASCAL FAR
ROEditWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bCWP = FALSE;


    switch( wMsg )
    {
      case WM_KEYDOWN:                /* trap all these messages */
      case WM_CHAR:
      case WM_MOUSEMOVE:
      case WM_LBUTTONDOWN:
      case WM_RBUTTONDOWN:
      case WM_RBUTTONUP:
```

(continued)

Figure 7-4. *continued*

```
        case WM_MBUTTONDOWN:
        case WM_MBUTTONUP:
          break;

        case WM_SETCURSOR:               /* trap client-area messages */
          bCWP = (LOWORD( lParam ) != HTCLIENT);
          break;

        default:
          bCWP = TRUE;
          break;
      }

      if( bCWP )
        lRVal = CallWindowProc( pDefEditWndFn, hWnd, wMsg, wParam, lParam );

      return lRVal;
  }
```

```
/************************************************************************
*                                                                       *
* WEP.C                                                                 *
*                                                                       *
* Exports:       WEP RESIDENTNAME                                       *
*                                                                       *
************************************************************************/

#define NOCOMM
#include    <windows.h>


/*** FUNCTION PROTOTYPES ***/

int PASCAL FAR      WEP( int );
```

(continued)

Figure 7-4. *continued*

```
/*********************************************************************
*                                                                    *
* WEP                                                                *
*                                                                    *
*********************************************************************/

int PASCAL FAR WEP( int nParam )
{
    return 1;
}
```

```
/*********************************************************************
*                                                                    *
* ROEDIT.H                                                           *
*                                                                    *
*********************************************************************/
/* defined in ROEDIT.C */
LONG PASCAL FAR ROEditWndFn( HWND, WORD, WORD, LONG );
```

```
;*********************************************************************
;                                                                    *
; ROEDIT.DEF module-definition file                                  *
;                                                                    *
;*********************************************************************

LIBRARY         ROEDIT
DESCRIPTION     'ROEDIT version 1.0'
EXETYPE         WINDOWS

CODE            LOADONCALL MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE SINGLE

HEAPSIZE        0

SEGMENTS        INIT_TEXT       PRELOAD DISCARDABLE
                WEP_TEXT        PRELOAD FIXED

EXPORTS         WEP             @1 RESIDENTNAME
                ROEditWndFn     @2
```

```
int PASCAL
WinMain( HANDLE hInst, HANDLE hPrevInst,
           LPSTR lpCmdLine, int nCmdShow )
{
    HWND    hWnd;
    MSG     msg;
    HANDLE  hDLL;

    /* load the library and verify the returned handle */
    hDLL = LoadLibrary( "ROEDIT.DLL" );
    if( hDLL < 32 )
      return 0;

    /* continue with the usual Windows processing */
    hWnd = Init( hInst, hPrevInst, nCmdShow );
    if( !hWnd )
      return 0;

    while( GetMessage( &msg, 0, 0, 0 ) )
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }

    /* free the library */
    FreeLibrary( hDLL );

    return msg.wParam;
}
```

Figure 7-5.
Loading and unloading the ROEDIT.DLL library in an application that uses the ROEdit *class.*

ROEdit is implemented by filtering keyboard and mouse messages before they are processed by the default edit-class window function. The function *ROEditWndFn* passes all other messages unchanged to the default edit-class window function, whose address is stored in a global variable by the *RegisterEditSubclass* function when the DLL is initialized.

The subclass window function, *ROEditWndFn*, looks straightforward, but actually a bit of magic is involved in its construction. Microsoft does not document how the default edit control's window function processes messages. In the case of *ROEditWndFn*, the magic lies in the ad hoc assumption that the way to create

a read-only control is to trap all keyboard and mouse input messages. This seems reasonable, but you must actually try it to be sure that the default edit-class window function is not carrying out some invisible, yet essential, action in response to one of the messages trapped in *ROEditWndFn*.

Owner-Draw Controls

Another technique for wringing new functionality out of the predefined control classes is to use owner-draw control styles. Owner-draw controls appeared in a somewhat different form in OS/2 Presentation Manager before they were introduced in Windows version 3.0. In Windows, owner-draw styles are supported for list-box, combo-box, and button controls, as well as for menus. The principle behind owner-draw controls is simple: Whenever Windows wants to change the appearance of a control with an owner-draw style, it sends a WM_DRAWITEM message to the control's owner—that is, its parent window. It is then up to the owner to redraw the control using the appropriate data, font, graphics, or colors.

Processing the WM_DRAWITEM Message

Owner-draw controls are valuable because they let an application determine both the appearance and the actual data displayed for each item in a control. An owner-draw control accomplishes this by sending a WM_DRAWITEM message to its owner for each item to be displayed. Each WM_DRAWITEM message is associated with a DRAWITEMSTRUCT data structure, which contains a 4-byte value (*itemData*) that identifies the data item to be displayed. How you use *itemData* depends on the application—*itemData* might contain a pointer to a string for text data, an RGB value for a color, or a handle to a GDI object. The DRAWITEMSTRUCT also contains a device-context handle (*hDC*) and an update rectangle (*rcItem*) so that the owner can update the appropriate part of the control's client area.

A key element in each WM_DRAWITEM message is a set of flags contained in two variables in the DRAWITEMSTRUCT structure. These flags indicate whether a data item needs to be repainted in its entirety or whether the data item should be repainted to reflect its selection status or a change in focus. The *itemAction* flags, shown in Figure 7-6 on the following page, inform the owner of an action that has changed the state of a data item. The *itemState* flags, shown in Figure 7-7 on the following page, describe the new state of the item. The owner should refer to the *itemAction* and *itemState* flags to redraw the data item so that its visual appearance reflects its new state.

Flag	Meaning
ODA_DRAWENTIRE	The entire data item needs to be redrawn.
ODA_FOCUS	The data item needs to be redrawn to indicate that the control has gained or lost the input focus.
ODA_SELECT	The data item needs to be redrawn to indicate that it has been selected or deselected.

Figure 7-6.
Flags defined in the itemAction *field in a DRAWITEMSTRUCT data structure.*

Flag	Meaning
ODS_CHECKED	The menu item is checked
ODS_DISABLED	The control is disabled
ODS_FOCUS	The control has the focus
ODS_GRAYED	The menu item is grayed
ODS_SELECTED	The data item is selected

Figure 7-7.
Flags defined in the itemState *field in a DRAWITEMSTRUCT data structure.*

The owner must also determine what data to display each time it processes a WM_DRAWITEM message. The *itemID* variable in the DRAWITEMSTRUCT structure contains an index value that indicates a data item's position in an owner-draw control or menu. For list-box and combo-box controls, the *itemData* value specifies the data item to display. An item's *itemData* value is determined at the time the item is first added to the control with a CB_ADDSTRING, LB_ADDSTRING, CB_INSERTSTRING, or LB_INSERTSTRING message.

It is easy to build an owner-draw control with an unusual appearance because the owner determines exactly how each data item is displayed. For example, you could build an owner-draw button that displays two custom bitmaps, one when the button is pressed and another when it is not. You could also design a read-only list box control by using an owner-draw list box that does not highlight the currently selected item.

Managing the Data

Owner-draw controls let you build list-box and combo-box controls that let the control's owner directly manage the list of data items displayed in the control. You can use this technique to design controls that display data other than null-terminated strings. You can also build controls that display more data than can be handled by Windows' built-in list box and combo box list-management routines.

Imagine, for example, that you want to use a list-box control to browse a database of 5,000 items. Without using the owner-draw style, 5,000 items probably represents more data than the list-box control can manage because the amount of memory the control can use to store its data is limited. (The limit is a little less than 64 KB in Windows 3.0.) With an owner-draw style, however, the control can avoid this memory limitation by storing 4-byte data-item identifiers instead of storing actual string data. When Windows sends the control's owner a WM_DRAWITEM message, it uses the *itemData* value in the accompanying DRAWITEMSTRUCT data structure to identify the string to display.

The source code for ODLB.EXE, in Figure 7-8, shows how this is done. Each time the application sends an LB_ADDSTRING message to the list box, it specifies a 4-byte numerical identifier instead of a string pointer. When the application processes WM_DRAWITEM messages, it uses the 4-byte identifiers to synthesize data strings on the fly. In a real Windows program, however, the strings might be obtained from an application-specific source such as a database-management system by using the *itemData* value to identify unique items. The sample application emulates the default appearance of a list-box control by using *PatBlt* and *DrawFocusRect* in response to the *itemAction* and *itemStatus* flags, but this too could be changed to meet the specific needs of a real application.

```
#***************************************************************************
#                                                                          *
# NMAKE description for ODLB.EXE                                           *
#                                                                          *
#***************************************************************************

.c.obj:
    cl /AM /c /G2sw /Osw /W4 /Zlp $*.c
```

Figure 7-8. *(continued)*
Source code for ODLB.EXE.

Figure 7-8. *continued*

```
ALL:          odlb.exe


odlb.obj:     odlb.c

odlb.exe:     odlb.obj odlb.def
    link /al:16 /nod /noe odlb, , , libw mlibcew, odlb.def
    rc odlb.exe
```

```
/*************************************************************************
 *                                                                       *
 * ODLB.C                                                                *
 *                                                                       *
 * Exports:       TopLevelWndFn                                          *
 *                                                                       *
 *************************************************************************/

#define     NOCOMM
#include    <windows.h>

#define     IDLISTBOX   0x1000
#define     IMAX        5000


/*** FUNCTION PROTOTYPES ***/

LONG PASCAL FAR TopLevelWndFn( HWND, WORD, WORD, LONG );

static HWND     Init( HANDLE, HANDLE, int );
static void     MsgCommand( HWND, WORD, LONG );
static void     MsgDrawItem( HWND, LPDRAWITEMSTRUCT );


/*** GLOBAL VARIABLES ***/

char    szTopLevelClass[] = "ModStat:TopLevel";
char    szAppTitle[] = "Owner-Draw ListBox";
HANDLE  hInstance = 0;
```

(continued)

Figure 7-8. *continued*

```
/*************************************************************************
*                                                                        *
* WinMain                                                                *
*                                                                        *
*************************************************************************/

int PASCAL
WinMain( HANDLE hInst, HANDLE hPrevInst, LPSTR lpszCmdLine, int nCmdShow )
{
    HWND    hWnd;
    MSG     msg;


    hWnd = Init( hInst, hPrevInst, nCmdShow );
    if( !hWnd )
      return 0;

    while( GetMessage( &msg, 0, 0, 0 ) )
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }

    return msg.wParam;
}


/*************************************************************************
*                                                                        *
* Init                                                                   *
*                                                                        *
*************************************************************************/

static HWND Init( HANDLE hInst, HANDLE hPrevInst, int nCmdShow )
{
    WNDCLASS    wc;
    HWND        hWnd, hListBox;
    int         n;
    LONG        lRVal;


    if( 0 == hPrevInst )
    {
      /* register the top-level window class */
      wc.lpszClassName    = szTopLevelClass;
      wc.hInstance        = hInst;
```

(continued)

Figure 7-8. *continued*

```
    wc.lpfnWndProc      = TopLevelWndFn;
    wc.hCursor          = LoadCursor( 0, IDC_ARROW );
    wc.hIcon            = LoadIcon( 0, IDI_APPLICATION );
    wc.lpszMenuName     = NULL;
    wc.hbrBackground    = COLOR_WINDOW+1;
    wc.style            = CS_HREDRAW | CS_VREDRAW;
    wc.cbClsExtra       = 0;
    wc.cbWndExtra       = 0;

    if( !RegisterClass( &wc ) )
      return 0;                        /* return 0 if unsuccessful */
  }

  /* save the current instance handle in a global variable */
  hInstance = hInst;

  /* create a top-level window */
  hWnd = CreateWindow( szTopLevelClass,
                       szAppTitle,
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, 0, 300, 320,
                       0,
                       0,
                       hInstance,
                       NULL );

  /* create an owner-draw listbox */
  hListBox = CreateWindow( "ListBox",
                       "",
                       WS_CHILD | WS_VISIBLE | WS_BORDER | WS_VSCROLL |
                         LBS_OWNERDRAWFIXED,
                       16, 16, 256, 256,
                       hWnd,
                       IDLISTBOX,
                       hInstance,
                       NULL );

  for( n=0; n<IMAX; n++ )
  {
    lRVal = SendMessage( hListBox, LB_ADDSTRING, 0, (LONG)n );
    if( lRVal < 0 )
      break;
  }
```

(continued)

Figure 7-8. *continued*

```
    ShowWindow( hWnd, nCmdShow );
    UpdateWindow( hWnd );

    return hWnd;
}


/************************************************************************
*                                                                       *
* TopLevelWndFn                                                         *
*                                                                       *
************************************************************************/

LONG PASCAL FAR
TopLevelWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bDWP = FALSE;


    switch( wMsg )
    {
      case WM_DRAWITEM:
        MsgDrawItem( hWnd, (LPDRAWITEMSTRUCT)lParam );
        break;

      case WM_SETFOCUS:
        SetFocus( GetDlgItem( hWnd, IDLISTBOX ) );
        break;

      case WM_DESTROY:
        PostQuitMessage( 0 );
        break;

      default:
        bDWP = TRUE;
        break;
    }

    if( bDWP )
      lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

    return lRVal;
}
```

(continued)

Figure 7-8. *continued*

```
/*********************************************************************
*                                                                    *
* MsgDrawItem                                                        *
*                                                                    *
*********************************************************************/

static void MsgDrawItem( HWND hWnd, LPDRAWITEMSTRUCT lpDIS )
{
    char    szBuf[32];


    /* if necessary, redraw the entire item */
    if( ODA_DRAWENTIRE & lpDIS->itemAction )
    {
      /* clear the item rectangle */
      PatBlt( lpDIS->hDC,
                     lpDIS->rcItem.left, lpDIS->rcItem.top,
                     lpDIS->rcItem.right - lpDIS->rcItem.left,
                     lpDIS->rcItem.bottom - lpDIS->rcItem.top,
                     PATCOPY );

      /* draw the output item */
      wsprintf( szBuf, "This is item %ld", lpDIS->itemData );
      TextOut( lpDIS->hDC,
                     lpDIS->rcItem.left, lpDIS->rcItem.top,
                     szBuf, lstrlen(szBuf) );

      /* invert the item rectangle if the item is selected */
      if( ODS_SELECTED & lpDIS->itemState )
        PatBlt( lpDIS->hDC,
                     lpDIS->rcItem.left, lpDIS->rcItem.top,
                     lpDIS->rcItem.right - lpDIS->rcItem.left,
                     lpDIS->rcItem.bottom - lpDIS->rcItem.top,
                     DSTINVERT );

      /* draw a focus rectangle if the item has the input focus */
      if( ODS_FOCUS & lpDIS->itemState )
        DrawFocusRect( lpDIS->hDC, &lpDIS->rcItem );
    }

    else
    {
      /* invert the item if the selection state is changing */
      if( ODA_SELECT & lpDIS->itemAction )
        PatBlt( lpDIS->hDC,
                     lpDIS->rcItem.left, lpDIS->rcItem.top,
```

(continued)

Figure 7-8. *continued*

```
                    lpDIS->rcItem.right - lpDIS->rcItem.left,
                    lpDIS->rcItem.bottom - lpDIS->rcItem.top,
                    DSTINVERT );

        /* redraw a focus rectangle if the focus state is changing */
        if( ODA_FOCUS & lpDIS->itemAction )
          DrawFocusRect( lpDIS->hDC, &lpDIS->rcItem );
    }
}
```

```
;************************************************************************
;                                                                       *
; ODLB.DEF module-definition file                                       *
;                                                                       *
;************************************************************************

NAME            ODLB
DESCRIPTION     'ODLB.EXE version 1.0'
EXETYPE         WINDOWS
STUB            'WINSTUB.EXE'

CODE            MOVEABLE LOADONCALL DISCARDABLE
DATA            MOVEABLE MULTIPLE PRELOAD

SEGMENTS        _TEXT   MOVEABLE PRELOAD DISCARDABLE

HEAPSIZE        1024
STACKSIZE       5120

EXPORTS         TopLevelWndFn
```

Using System Commands

The WM_SYSCOMMAND message is a general-purpose message that Windows uses to change a window's size or position. Windows sends a window a WM_SYSCOMMAND message when you select the Restore, Move, Size, Minimize, Maximize, or Close command from the window's system menu. Windows also sends WM_SYSCOMMAND when you click the mouse in the window's non-client area or when you use key combinations such as Alt-Spacebar to select a menu or Alt-Esc to switch between application windows.

Message Subtype (wParam)	Corresponding System-Menu Selection, Keypress, or Mouse Action	Parameters
SC_SIZE	Size	Bits 0-3 of *wParam*: 0: use keyboard to size the window 1: mouse on left border 2: mouse on right border 3: mouse on top border 4: mouse on upper left corner 5: mouse on upper right corner 6: mouse on bottom border 7: mouse on lower left corner 8: mouse on lower right corner
SC_MOVE	Move	Bits 0-3 of *wParam*: 0: use keyboard to move the window 2: use mouse to move the window
SC_MAXIMIZE	Maximize	Bits 0-3 of *wParam*: 2: mouse double-click on title bar
SC_MINIMIZE	Minimize	
SC_NEXTWINDOW	Alt-Esc (activate next window)	
SC_CLOSE	Close	
SC_VSCROLL	Click on vertical scroll bar	
SC_HSCROLL	Click on horizontal scroll bar	
SC_MOUSEMENU	Click on control-menu box	Bits 0-3 of *wParam*: 3: mouse on control-menu box
SC_KEYMENU	Alt-Spacebar	
SC_RESTORE	Restore	Bits 0-3 of *wParam*: 2: mouse double-click on title bar
SC_TASKLIST	Ctrl-Esc (display "Task List" window)	*lParam*: Cursor position as MAKELONG(x,y) or 0L

Figure 7-9.
WM_SYSCOMMAND message types in wParam. *Bits 4-15 of* wParam *contain the message subtype. Bits 0-3 of* wParam *can contain a mouse hit-test code.*

WM_SYSCOMMAND Subtypes

Windows 3.0 supports 13 message subtypes for WM_SYSCOMMAND. Each subtype corresponds to a different action, as shown in Figure 7-9. The subtype values occupy the high-order 12 bits of *wParam*. In most cases, the low-order 4 bits of *wParam* are unused. However, when a WM_SYSCOMMAND message is generated by clicking the mouse on a window's non-client area, the low-order 4 bits contain a non-zero hit-test code that specifies where the mouse was clicked. To determine the subtype of a WM_SYSCOMMAND message, examine only the 12 high-order bits of *wParam*, as follows:

```
if( SC_SIZE == (wParam & 0xFFF0) ) ...
```

Filtering WM_SYSCOMMAND Messages

Most window functions pass WM_SYSCOMMAND messages to the default window function, *DefWindowProc*, which does the necessary moving, resizing, and closing. If you process WM_SYSCOMMAND in a window function, you can alter the functions of a window's system menu. For example, you can create a window that is always maximized by filtering the WM_SYSCOMMAND messages that correspond to the Minimize, Restore, Size, and Move system menu commands, as shown in Figure 7-10.

```
LONG PASCAL FAR
MaxedWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    BOOL  bDWP = FALSE;
    LONG  lRVal = 0L;


    switch( wMsg )
    {
      case WM_SYSCOMMAND:
        switch( wParam & 0xFFF0 )
        {
          case SC_SIZE:
          case SC_MOVE:
          case SC_MINIMIZE:
          case SC_RESTORE:
            bDWP = FALSE;
            break;
```

Figure 7-10. *(continued)*
A window function that filters WM_SYSCOMMAND messages to keep a window maximized.

Figure 7-10. *continued*

```
            default:
              bDWP = TRUE;
              break;
          }
          break;

          /* (other message processing) */

      }

      if( bDWP )
        lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

      return lRVal;
  }
```

Emulating System Commands

You can emulate the functions of a window's system menu by synthesizing WM_SYSCOMMAND messages and sending them to *DefWindowProc*. To do this, send or post a WM_SYSCOMMAND message to the window, with *wParam* set to the message subtype that corresponds to the action you want Windows to carry out. For the SC_MOVE and SC_SIZE subtypes, you should also set bits 0 through 3 of *wParam* to indicate whether to emulate a mouse action or a keyboard action.

Imagine, for example, that you want to create a child-window control that can be moved within its parent's client area without using a system menu, a title bar, or a thick frame. The way to do this is to post WM_SYSCOMMAND messages with the SC_MOVE type. If bits 0 through 3 contain 0 as they do in the following function call, the default window function displays a four-arrow cursor and uses keyboard input to move the window:

```
PostMessage( hWnd, WM_SYSCOMMAND, SC_MOVE, 0L );
```

If bits 0 through 3 of *wParam* contain the value 2, the default window function processes the message as if it had been generated by clicking on a window's title bar so that the user can move the window by dragging it with the mouse:

```
PostMessage( hWnd, WM_SYSCOMMAND, SC_MOVE | 0x0002, 0L );
```

A good time to post this message is in response to a WM_SETCURSOR or WM_LBUTTONDOWN message, as you will see in the next source-code example.

Customizing the Non-Client Area

Few Windows programs contain windows that paint their own non-client areas. For a window with a standard non-client area, the default window function *DefWindowProc* processes the WM_SETCURSOR messages that correspond to mouse activity in the window's non-client area, sends WM_SYSCOMMAND messages as needed for sizing and moving the window, and responds to WM_NCPAINT by repainting the window's non-client area. To customize a window's non-client area, your window function must process non-client area messages that would otherwise be handled by *DefWindowProc.*

The source code for ROUND.EXE, shown in Figure 7-11, illustrates some of the techniques involved in working with a window's non-client area. The program creates a set of nine round child windows that can be moved by dragging with the mouse, as shown in Figure 7-12 on page 243. The round appearance is an illusion—the windows are actually rectangular windows with a rectangular non-client area large enough to contain a circular border. To support the illusion, the window function *RoundWndFn* hit-tests and paints its non-client area by processing WM_NCCALCSIZE, WM_NCPAINT, and WM_SETCURSOR messages. In a window that did not manage its own non-client area, these messages would normally be passed through to *DefWindowProc.*

```
#*************************************************************************
#                                                                        *
# NMAKE description for ROUND.EXE                                        *
#                                                                        *
#*************************************************************************

.c.obj:
    cl /AM /c /G2sw /Osw /W4 /Zlp $*.c


ALL:            round.exe

round.obj:      round.c round.h

wndround.obj:   wndround.c round.h


round.res:      round.rc round.ico round.h
    rc /r round.rc
```

Figure 7-11. *(continued)*
Source code for ROUND.EXE.

Figure 7-11. *continued*

```
round.exe:      round.obj wndround.obj  round.res round.def
    link /al:16 /nod /noe round wndround, , , libw mlibcew, round.def
    rc round.res
```

```
/************************************************************************
*                                                                       *
* ROUND.C                                                               *
*                                                                       *
* Exports:      TopLevelWndFn                                           *
*                                                                       *
************************************************************************/

#define NOCOMM
#include    <windows.h>
#include    "round.h"


/*** FUNCTION PROTOTYPES ***/

LONG PASCAL FAR    TopLevelWndFn( HWND, WORD, WORD, LONG );

static HWND        Init( HANDLE, HANDLE, int );


/*** GLOBAL VARIABLES ***/

char        szTopLevelClass[] = "Round:TopLevel";
char        szRoundWndClass[] = "Round:Child";


/************************************************************************
*                                                                       *
* WinMain                                                               *
*                                                                       *
************************************************************************/

int PASCAL
WinMain( HANDLE hInst, HANDLE hPrevInst, LPSTR lpszCmdLine, int nCmdShow )
{
    HWND    hWnd;
    MSG     msg;
```

(continued)

Figure 7-11. *continued*

```
    hWnd = Init( hInst, hPrevInst, nCmdShow );
    if( !hWnd )
      return 0;

    while( GetMessage( &msg, 0, 0, 0 ) )
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }

    return msg.wParam;
}


/*************************************************************************
 *                                                                       *
 * Init                                                                  *
 *                                                                       *
 *************************************************************************/

static HWND Init( HANDLE hInst, HANDLE hPrevInst, int nCmdShow )
{
    WNDCLASS    wc;
    HWND        hWnd, hChild;
    int         n;
    char        szTitle[6];


    if( 0 == hPrevInst )
    {
      /* register the top-level window class */
      wc.lpszClassName    = szTopLevelClass;
      wc.hInstance        = hInst;
      wc.lpfnWndProc      = TopLevelWndFn;
      wc.hCursor          = LoadCursor( 0, IDC_ARROW );
      wc.hIcon            = LoadIcon( hInst, "RoundIcon" );
      wc.lpszMenuName     = NULL;
      wc.hbrBackground    = COLOR_WINDOW+1;
      wc.style            = CS_HREDRAW | CS_VREDRAW;
      wc.cbClsExtra       = 0;
      wc.cbWndExtra       = 0;

      if( !RegisterClass( &wc ) )
        return 0;                       /* return 0 if unsuccessful */
```

(continued)

Figure 7-11. *continued*

```
    /* register the round window class */
    wc.lpszClassName    = szRoundWndClass;
    wc.hInstance        = hInst;
    wc.lpfnWndProc      = RoundWndFn;
    wc.hCursor          = LoadCursor( 0, IDC_ARROW );
    wc.hIcon            = 0;
    wc.lpszMenuName     = NULL;
    wc.hbrBackground    = 0;
    wc.style            = 0;
    wc.cbClsExtra       = 0;
    wc.cbWndExtra       = 0;

    if( !RegisterClass( &wc ) )
      return 0;                          /* return 0 if unsuccessful */
  }

  /* create the top-level window */
  hWnd = CreateWindow( szTopLevelClass,
                       "Round Windows",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, 0, 220, 240,
                       0,
                       0,
                       hInst,
                       NULL );

  /* create nine round child windows */
  for( n=0; n<9; n++ )
  {
    hChild = CreateWindow( szRoundWndClass,
                           "",
                           WS_CHILD | WS_VISIBLE,
                           (n%3)*64+12, (n/3)*64+12, 60, 60,
                           hWnd,
                           0,
                           hInst,
                           NULL );

    wsprintf( szTitle, "%04X", hChild );
    SetWindowText( hChild, szTitle );
  }
```

(continued)

Figure 7-11. *continued*

```
    /* display the top-level window */
    ShowWindow( hWnd, nCmdShow );
    UpdateWindow( hWnd );

    return hWnd;
}


/************************************************************************
*                                                                       *
* TopLevelWndFn                                                         *
*                                                                       *
************************************************************************/

LONG PASCAL FAR
TopLevelWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bDWP = FALSE;


    switch( wMsg )
    {
      case WM_DESTROY:
        PostQuitMessage( 0 );
        break;

      default:
        bDWP = TRUE;
        break;
    }

    if( bDWP )
      lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

    return lRVal;
}
```

(continued)

Figure 7-11. *continued*

```
/****************************************************************************
 *                                                                          *
 * WNDROUND.C                                                               *
 *                                                                          *
 * Exports:      RoundWndFn                                                 *
 *                                                                          *
 ****************************************************************************/


#define     NOCOMM
#include    <windows.h>
#include    "round.h"


/*** FUNCTION PROTOTYPES ***/

static void MsgPaint( HWND );
static void MsgNCPaint( HWND );
static void DrawRoundFrame( HWND );


/*** GLOBAL VARIABLES ***/

static DWORD dwBorderColor = RGB(0xFF,0x00,0x00);    /* red */
static DWORD dwFgdColor    = RGB(0xFF,0xFF,0xFF);    /* white */
static DWORD dwBkgdColor   = RGB(0x00,0x00,0xFF);    /* blue */


/****************************************************************************
 *                                                                          *
 * RoundWndFn                                                               *
 *                                                                          *
 ****************************************************************************/

LONG PASCAL FAR
RoundWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bDWP = FALSE;
    RECT    rect;
    HWND    hParent;
    int     nDiff;
```

(continued)

Figure 7-11. *continued*

```
    switch( wMsg )
    {
      case WM_NCCALCSIZE:
        nDiff = -3*GetSystemMetrics( SM_CXFRAME );
        InflateRect( (LPRECT)lParam, nDiff, nDiff );
        break;

      case WM_PAINT:
        MsgPaint( hWnd );
        break;

      case WM_ERASEBKGND:
        lRVal = TRUE;
        break;

      case WM_NCPAINT:
        MsgNCPaint( hWnd );
        break;

      case WM_SETCURSOR:
        if( HTCLIENT != LOWORD(lParam) )
        {
          switch( HIWORD(lParam) )
          {
            case WM_LBUTTONDOWN:
              PostMessage( hWnd, WM_SYSCOMMAND, SC_MOVE | 0x0002, 0L );
              break;

            default:
              SetCursor( LoadCursor( 0, IDC_CROSS ) );
              break;
          }
        }

        else
          bDWP = TRUE;

        break;

      case WM_MOVE:
        GetWindowRect( hWnd, &rect );
        hParent = GetParent( hWnd );
        ScreenToClient( hParent, (LPPOINT)&rect.left );
        ScreenToClient( hParent, (LPPOINT)&rect.right );
        InvalidateRect( hParent, &rect, TRUE );
        break;
```

(continued)

Figure 7-11. *continued*

```
      default:
        bDWP = TRUE;
        break;
    }

    if( bDWP )
      lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

    /* display the most recently moved child window last */
    if( (WM_SYSCOMMAND == wMsg) && ((SC_MOVE | 0x0002) == wParam) )
      SetWindowPos( hWnd, 1, 0, 0, 0, 0,
                    SWP_NOMOVE | SWP_NOSIZE | SWP_NOREDRAW );

    return lRVal;
}


/*************************************************************************
 *                                                                       *
 * MsgPaint                                                              *
 *                                                                       *
 *************************************************************************/

static void MsgPaint( HWND hWnd )
{
    HDC         hDC;
    PAINTSTRUCT ps;
    RECT        rect;
    char        szText[6];


    hDC = BeginPaint( hWnd, &ps );

    /* show the window text */
    GetWindowText( hWnd, szText, sizeof szText );
    GetClientRect( hWnd, &rect );

    SetTextColor( hDC, dwFgdColor );
    SetBkColor( hDC, dwBkgdColor );
    DrawText( hDC, szText, lstrlen(szText), &rect,
              DT_SINGLELINE | DT_CENTER | DT_VCENTER );

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

(continued)

Figure 7-11. *continued*

```
/*********************************************************************
 *                                                                   *
 * MsgNCPaint                                                        *
 *                                                                   *
 *********************************************************************/

static void MsgNCPaint( HWND hWnd )
{
    DrawRoundFrame( hWnd );
    InvalidateRect( hWnd, NULL, TRUE );
    UpdateWindow( hWnd );
}


/*********************************************************************
 *                                                                   *
 * DrawRoundFrame                                                    *
 *                                                                   *
 * Note: This routine also fills the client area                     *
 *       with the background brush.                                  *
 *                                                                   *
 *********************************************************************/

static void DrawRoundFrame( HWND hWnd )
{
    HDC     hDC;
    RECT    rect;
    HBRUSH  hBrush;
    HPEN    hPen;


    hDC = GetWindowDC( hWnd );
    GetWindowRect( hWnd, &rect );
    OffsetRect( &rect, -rect.left, -rect.top );

    /* draw a non-client ellipse */
    hBrush = CreateSolidBrush( dwBkgdColor );
    hBrush = SelectObject( hDC, hBrush );
    hPen = CreatePen( PS_INSIDEFRAME,
                      GetSystemMetrics( SM_CXFRAME ),
                      dwBorderColor );
    hPen = SelectObject( hDC, hPen );

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

(continued)

Figure 7-11. *continued*

```
    hPen = SelectObject( hDC, hPen );
    DeleteObject( hPen );
    hBrush = SelectObject( hDC, hBrush );
    DeleteObject( hBrush );

    ReleaseDC( hWnd, hDC );
}
```

```
/***********************************************************************
*                                                                      *
* ROUND.RC resource script                                             *
*                                                                      *
***********************************************************************/

/* icons */
RoundIcon       ICON    round.ico
```

```
/************************************************************************
*                                                                       *
* ROUND.H                                                               *
*                                                                       *
************************************************************************/

/* defined in WNDROUND.C */
LONG PASCAL FAR     RoundWndFn( HWND, WORD, WORD, LONG );
```

```
;***********************************************************************
;                                                                      *
; ROUND.DEF module-definition file                                     *
;                                                                      *
;***********************************************************************

NAME            ROUND
DESCRIPTION     'ROUND.EXE version 1.0'
EXETYPE         WINDOWS
```

(continued)

Figure 7-11. *continued*

```
STUB            'WINSTUB.EXE'

CODE            MOVEABLE LOADONCALL DISCARDABLE
DATA            MOVEABLE MULTIPLE PRELOAD

SEGMENTS        _TEXT   MOVEABLE PRELOAD DISCARDABLE

HEAPSIZE        512
STACKSIZE       5120

EXPORTS         TopLevelWndFn
                RoundWndFn
```

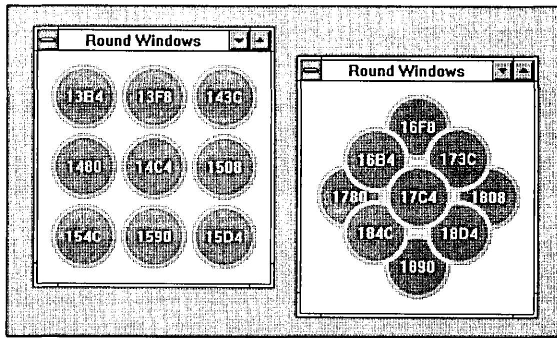


Figure 7-12.
Two instances of ROUND.EXE. Each child window displays its window handle.

Client-Area Size

Windows sends WM_NCCALCSIZE to a window function to obtain the boundaries of the window's client area. The non-client area lies between the rectangle that defines the outside of the window—that is, the window rectangle—and the client area. In response to WM_NCCALCSIZE, *DefWindowProc* uses the window's style to compute a client area that accommodates the window's border, menus, scroll bars, and any other non-client area elements. However, in this sample program, *RoundWndFn* processes WM_NCCALCSIZE explicitly by computing a client area small enough to leave room for an ellipse with a fat border in the non-client area.

Painting

To paint the non-client area, *RoundWndFn* processes the WM_NCPAINT message. This action consists only of drawing an ellipse whose border lies within the window's non-client area. *RoundWndFn* takes a shortcut by relying on the *Ellipse* function (which is called by *DrawRoundFrame*) to paint the window's client-area background at the time WM_NCPAINT is processed, so the WM_ERASEBKGND message can be trapped and not processed. In a different application, however, you may need to paint the client-area background by passing WM_ERASEBKGND to *DefWindowProc* or by processing the message explicitly.

Hit Testing

The WM_SETCURSOR case in *RoundWndFn* carries out hit testing for the round child windows. When a WM_SETCURSOR message indicates that the user has pressed the left mouse button in a child window's non-client area, *RoundWndFn* sends the child window a WM_SYSCOMMAND message that causes Windows to let the user move the window with the mouse.

In response to other WM_SETCURSOR messages in the non-client area, *RoundWndFn* displays a cursor whose shape depends on its position in the child window. The hit-test code in the low-order word of *lParam* indicates whether the cursor lies in the window's client area. If the cursor does not lie in the window's client area, *RoundWndFn* assumes it is located over the non-client area and calls *SetCursor* to display a crosshair cursor. If the cursor does lie in the client area, *RoundWndFn* calls *DefWindowProc,* which displays the class cursor, the default arrow.

If you use the crosshair cursor to trace the outline of the non-client area of each round child window, you will see that the non-client area is not round but rectangular. Therefore, the hit testing in *RoundWndFn* isn't really accurate. To do it right, you would need to examine the cursor coordinates each time the left mouse button is pressed in the non-client area and determine whether the cursor lies not only within the window's non-client area but also within the elliptical window border.

Overlapping and Clipping

Although the child windows in ROUND.EXE appear round, Windows overlaps them and clips them as rectangular windows. You can demonstrate this by adding the WS_CLIPSIBLINGS style to the *CreateWindow* call that creates the

round child windows. When you do this, you can see how Windows clips each child window by using the child's window rectangle. The application supports the illusion that the child windows are round by explicitly repainting child windows rather than using the WS_CLIPSIBLINGS style. For this strategy to work, however, *RoundWndFn* must call *SetWindowPos* to place the most-recently accessed child window at the end of the window manager's list of child windows. This causes the most-recently accessed child window to be painted last.

This seems straightforward, but it leads to subtle problems with hit testing and painting overlapping windows. You can verify this by experimenting with ROUND.EXE. You could attempt to remedy these problems by explicitly enumerating the window manager's list of child windows and by doing your own hit testing and painting, but it is best to avoid such problems altogether by using only rectangular windows and letting the window manager do the work.

Handling Asynchronous Events

In any microcomputer, a variety of events occur asynchronously—that is, without being synchronized with whatever the CPU happens to be doing at the moment the event occurs. Events such as hardware timer ticks, key presses, mouse movements, and receipt of data through a serial communications port almost always occur at times when the CPU is not idle. Such events are typically signaled through hardware or software interrupts. The CPU processes an interrupt by transferring control to a special-purpose function called an interrupt handler, which carries out some specific action in response to the event.

In Windows, nearly all asynchronous events are managed by interrupt handlers contained in device drivers installed at the time the Windows environment is initialized. For example, KEYBOARD.DRV, MOUSE.DRV, and COMM.DRV contain interrupt handlers for the interrupts generated by keyboard, mouse, and serial communications activity. Although such device drivers take care of the vast majority of asynchronous events, there are situations in which an application or a DLL must handle asynchronous events on its own. A typical example is a DLL that uses the NetBIOS local-area network communications interface.

NetBIOS in Windows

NetBIOS is a protocol developed by IBM for communicating on local-area networks. NetBIOS is supported by a number of software vendors, including IBM, Microsoft, and Novell. Programs running in MS-DOS, OS/2, and Windows can use NetBIOS to communicate across local-area networks. The next few para-

graphs describe how to access NetBIOS in a Windows program. If you aren't already familiar with the NetBIOS protocol, you can find it described in IBM's LAN Technical Reference manual (document #SC30-3383) as well as in a number of books on local-area network programming.

To execute a NetBIOS command, you first initialize a data structure called a Network Control Block (NCB) with a predefined set of parameters, as shown in Figure 7-13. You then pass the NCB's address to NetBIOS, which carries out the command. In a non-Windows program running under MS-DOS, you call NetBIOS by placing the address of an NCB in registers ES and BX and executing software interrupt 5CH. In Windows, you call an API function, *NetBIOSCall*, instead of executing the software interrupt, as shown in Figure 7-14. The result is the same: NetBIOS carries out the command and returns control to the calling program.

```
typedef struct          /* NetBIOS control block */
{
  BYTE    cCommand;       /* command code */
  BYTE    cRetcode;       /* return code */
  BYTE    cLSN;           /* local session number */
  BYTE    cNum;           /* number of name in local name table */
  LPSTR   lpBuffer;       /* message buffer address */
  WORD    wLength;        /* message buffer length */
  BYTE    cCallName[16];  /* local or remote NetBIOS name */
  BYTE    cName[16];      /* local NetBIOS name */
  BYTE    cRTO;           /* receive timeout count */
  BYTE    cSTO;           /* send timeout count */
  FARPROC fnPost;         /* address of post routine */
  BYTE    cAdapterNum;    /* 0=1st adapter; 1=2nd adapter */
  BYTE    cCmdCplt;       /* command status */
  BYTE    cReserved[14];  /* reserved area */
}
        NCB;
```

Figure 7-13.
A C-language declaration for a NetBIOS Network Control Block (NCB).

Because network data transmissions can take several seconds to execute, NetBIOS lets an application initiate network transactions without waiting for them to complete. When a prolonged network transaction completes, NetBIOS notifies the program by calling a post routine, a user-defined function whose address is passed to NetBIOS by the program when it initiates a transaction.

```
;
; _NETInt
;
; Caller:
;  int PASCAL FAR _NETInt( NCB FAR * lpNCB );
;

            EXTRN    NETBIOSCALL:far

            PUBLIC   _NETINT
_NETINT     PROC     far

            push     bp
            mov      bp,sp

            les      bx,[bp+6]    ; ES:BX -> NCB
            call     NETBIOSCALL

            xor      ah,ah        ; AX = return value

            pop      bp
            ret      4

_NETINT     ENDP
```

Figure 7-14.
Using NetBIOSCall *in Windows version 3.*

Consider, for example, how a Windows program might use NetBIOS to receive a packet of data from another computer on a local-area network. The program initiates the process of receiving a packet of data by calling the NetBIOS NCB.RECEIVE command. The NCB used for this call contains the address of a post routine and the address of a buffer to be used to contain the received data. NetBIOS processes the NCB.RECEIVE command by starting to wait for a data packet. The call to NetBIOS returns immediately, so the program can continue executing while NetBIOS waits for data to arrive. Later, when the data has been received, NetBIOS calls the post routine to notify the program that the received data is available.

An Asynchronous-Event Handler

The key to writing a NetBIOS post routine is to realize that the event that triggers a call to the routine occurs asynchronously, outside of Windows' multitasking and message-processing mechanisms. In order for the post routine to work, it must notify a Windows program that an event has occurred without disrupting

program task management or message flow. To accomplish this, the post routine must place a message into an application's message queue with a call to *PostMessage* or *PostAppMessage.*

The source code in Figure 7-15 contains a post routine, *_NCBPost,* that demonstrates how this can be done. When NetBIOS calls *_NCBPost,* registers ES:BX contain a pointer to the NCB whose processing has just been completed. The *_NCBPost* routine passes this pointer to *PostNCBMessage,* which uses the pointer to obtain a window handle from a static table. *PostNCBMessage* then calls *PostMessage,* which places a user-defined message into the appropriate message queue. In this way, the asynchronously executed post routine notifies an application that NetBIOS processing for the NCB has completed.

```
;
; This routine calls a C function defined as:
;  void PASCAL FAR PostNCBMsg(NCB FAR * lpNCB, int nCompletionCode)
;  {
;    PostMessage( ... );
;  }
;

        EXTRN   POSTNCBMSG:far

        PUBLIC  _NCBPOST
_NCBPOST    PROC    far         ; at entry: AL = completion code
                                ;           AH = 0
                                ;           ES:BX->NCB

        pusha                   ; save all registers

        push    es              ; push NCB address
        push    bx
        push    ax              ; push completion code

        call    POSTNCBMSG      ; call C routine
                                ;  to call PostMessage

        popa                    ; restore all registers
        iret                    ; return to NetBIOS

_NCBPOST    ENDP
```

Figure 7-15.
Source code for a NetBIOS post routine for Windows version 3.

The reason this design works is that *PostMessage* is re-entrant—that is, if NetBIOS happens to call *_NCBPost* at a moment when Windows is executing *PostMessage*, *_NCBPost*'s own call to *PostMessage* will execute properly without disrupting the previous call to *PostMessage*. *PostMessage* and *PostAppMessage* are the only functions in the Windows API that are guaranteed to be re-entrant, so they are the only API functions that are safe to call in an asynchronously executed routine such as *_NCBPost*. This is the technique to use when you write your own asynchronous-event handler.

A Quick Exit

In some settings, a Windows user may find it convenient to be able to exit quickly from the Windows environment without switching to a shell application such as the Program Manager. To do this, use the *ExitWindows* API function to terminate all applications cleanly and exit from the Windows environment. The sample program in Figure 7-16 shows how to use *ExitWindows* in this way. The application displays an icon that you can click with the mouse or select by pressing Alt-Spacebar to terminate a Windows session.

```
#************************************************************************
#                                                                       *
# NMAKE description for WINEXIT.EXE                                     *
#                                                                       *
#************************************************************************

.c.obj:
    cl /AM /c /G2sw /Osw /W4 /Zlp $*.c

ALL:            winexit.exe

winexit.obj:    winexit.c

winexit.res:    winexit.rc winexit.ico
    rc /r winexit.rc

winexit.exe:    winexit.obj winexit.res winexit.def
    link /al:16 /nod /noe winexit, , , libw mlibcew, winexit.def
    rc winexit.res
```

Figure 7-16.
Source code for WINEXIT.EXE

(continued)

Figure 7-16. *continued*

```
/****************************************************************************
 *                                                                          *
 * WINEXIT.C                                                                *
 *                                                                          *
 * Exports:  TopLevelWndFn                                                  *
 *                                                                          *
 ****************************************************************************/

#define     NOCOMM
#include    <windows.h>


/*** FUNCTION PROTOTYPES ***/

LONG PASCAL FAR TopLevelWndFn( HWND, WORD, WORD, LONG );

static HWND     Init( HANDLE, HANDLE, int );
static BOOL     QueryExit( HWND );


/*** GLOBAL VARIABLES ***/

char        szTopLevelClass[] = "WinExit:TopLevel";


/****************************************************************************
 *                                                                          *
 * WinMain                                                                  *
 *                                                                          *
 ****************************************************************************/

int PASCAL
WinMain( HANDLE hInst, HANDLE hPrevInst, LPSTR lpszCmdLine, int nCmdShow )
{
    HWND    hWnd;
    MSG     msg;


    hWnd = Init( hInst, hPrevInst, nCmdShow );
    if( !hWnd )
      return 0;
```

(continued)

Figure 7-16. *continued*

```
    while( GetMessage( &msg, 0, 0, 0 ) )
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }

    return msg.wParam;
}


/************************************************************************
*                                                                       *
* Init                                                                  *
*                                                                       *
************************************************************************/

static HWND Init( HANDLE hInst, HANDLE hPrevInst, int nCmdShow )
{
    WNDCLASS    wc;
    HWND        hWnd = 0;


    /* allow only one instance */
    if( hPrevInst )
      return 0;

    /* register the top-level window */
    wc.lpszClassName    = szTopLevelClass;
    wc.hInstance        = hInst;
    wc.lpfnWndProc      = TopLevelWndFn;
    wc.hCursor          = LoadCursor( 0, IDC_ARROW );
    wc.hIcon            = LoadIcon( hInst, "TopLevelIcon" );
    wc.lpszMenuName     = NULL;
    wc.hbrBackground    = COLOR_WINDOW+1;
    wc.style            = 0L;
    wc.cbClsExtra       = 0;
    wc.cbWndExtra       = 0;

    if( !RegisterClass( &wc ) )
      return 0;

    /* create the top-level window */
    hWnd = CreateWindow( szTopLevelClass,
                         "Windows Exit",
                         WS_OVERLAPPEDWINDOW | WS_ICONIC,
                         0, 0, 0, 0,
```

(continued)

Figure 7-16. *continued*

```
                        0,
                        0,
                        hInst,
                        NULL );

    /* put window on screen */
    ShowWindow( hWnd, SW_SHOWMINNOACTIVE );
    UpdateWindow( hWnd );

    return hWnd;
}


/***********************************************************************
*                                                                      *
* TopLevelWndFn                                                        *
*                                                                      *
***********************************************************************/

LONG PASCAL FAR
TopLevelWndFn( HWND hWnd, WORD wMsg, WORD wParam, LONG lParam )
{
    LONG    lRVal = 0L;
    BOOL    bDWP = FALSE;


    switch( wMsg )
    {
      case WM_QUERYOPEN:
        break;

      case WM_DESTROY:
        PostQuitMessage( 0 );
        break;

      case WM_NCLBUTTONDOWN:
        QueryExit( hWnd );
        break;

      case WM_SYSCHAR:
        switch( wParam )
        {
          case VK_SPACE:
            if( 0x20000000L & lParam )  /* Alt-Spacebar */
              QueryExit( hWnd );
            break;
```

(continued)

Figure 7-16. *continued*

```
          case VK_RETURN:                    /* Enter */
            QueryExit( hWnd );
            break;
        }

        break;

      default:
        bDWP = TRUE;
        break;
    }

    if( bDWP )
      lRVal = DefWindowProc( hWnd, wMsg, wParam, lParam );

    return lRVal;
}


/*************************************************************************
 *                                                                       *
 * QueryExit                                                             *
 *                                                                       *
 *************************************************************************/

static BOOL QueryExit( HWND hWnd )
{
    BOOL   bRVal = TRUE;


    if( IDOK == MessageBox( hWnd,
                            "This will end your Windows session.",
                            "End Session",
                            MB_OKCANCEL | MB_ICONEXCLAMATION |
                              MB_SYSTEMMODAL ) )
    {
      bRVal = ExitWindows( 0L, 0 );
    }

    return bRVal;
}
```

(continued)

Figure 7-16. *continued*

```
/*************************************************************************
*                                                                        *
* WINEXIT.RC resource script                                             *
*                                                                        *
*************************************************************************/

/* icons */
TopLevelIcon      ICON    winexit.ico
```

```
;*************************************************************************
;                                                                        *
; WINEXIT.DEF module-definition file                                     *
;                                                                        *
;*************************************************************************

NAME            WINEXIT
DESCRIPTION     'WINEXIT.EXE version 1.0'
EXETYPE         WINDOWS

STUB            'WINSTUB.EXE'

CODE            MOVEABLE PRELOAD DISCARDABLE
DATA            MOVEABLE SINGLE PRELOAD

HEAPSIZE        512
STACKSIZE       5120

EXPORTS         TopLevelWndFn
```

Appendix

A Windows Programming Glossary

386 enhanced mode. *See* Enhanced mode.

API. *See* Application program interface.

Application program interface (API). Windows' API contains more than 600 function calls that support I/O, manage memory resources, allow multitasked applications to execute cooperatively, and provide a variety of other services.

Atom. In Windows, an atom is a data string identified by a unique integer value. Windows provides an atom-management API for atoms stored in a program's local heap as well as for atoms stored globally.

Automatic data segment. *See* Default data segment.

DDE. Dynamic Data Exchange.

Default data segment. A data segment, associated with an application or a dynamic link library, that can contain static data, a stack, and a local heap. In an assembly-language program, the default data segment is named _DATA and is part of group DGROUP. In a C program, the Microsoft C compiler takes care of naming and grouping the default data segment.

Dereference. To convert an indirect reference to a direct reference. For example, the Windows API function *GlobalLock* dereferences a memory handle by returning a physical address that can be used to reference the memory block identified by the handle.

Device. A video display, printer, disk drive, or other hardware used for input or output.

Device driver. A low-level software interface to a specific hardware device. In Windows, device drivers are usually implemented as DLLs with the filename extension .DRV. A typical Windows installation includes device drivers that support the video display, mouse, keyboard, serial communications port, and printer.

DLL. *See* Dynamic link library.

Dynamic link library (DLL). A windows module containing executable code and data that can be referenced by Windows applications or by functions in other DLLs. Functions and data in a DLL are dynamically linked to the functions that reference them—that is, the links are made at runtime when the library is loaded or later.

Enhanced mode. A Windows operating mode in which Windows' memory manager uses virtual memory to manage the global heap. In enhanced mode, Windows applications run in 16-bit protected mode and non-Windows applications run in virtual 8086 mode. Also, in enhanced mode, Windows applications share access to hardware devices and local area networks through virtual devices that run in 32-bit protected mode. Supported on 80386 and 80486 microprocessors.

Entry point. The logical start of an executable function. In Windows, function entry points can be identified by ordinal number as well as by name.

Expanded memory. *See* LIM EMS.

Extended memory. Memory whose physical addresses lie above 1 MB. Extended memory is addressable only in protected mode on 80286, 80386, or 80486 microprocessors. Windows uses extended memory in standard and enhanced CPU modes.

Far pointer. A far pointer specifies a memory address as a 32-bit value that designates both a particular segment in memory and an offset within that segment. *See also* Near pointer.

GDI. *See* Graphics Device Interface.

Graphics Device Interface (GDI). The Graphics Device Interface is a set of Windows API functions that support graphical output to the video display and to a variety of printers and plotters.

Handle. An arbitrary numeric value assigned by Windows to let an application identify a certain item. Windows uses handles to identify a variety of items, including modules, instances, windows, memory blocks, and GDI objects.

Hash table. A table of values in which data items are scattered according to a formula that allows the table to be quickly searched for a particular value. Windows uses hash tables to store data such as the string values associated with local or global atoms.

Heap. An area of memory reserved for dynamic allocation of memory blocks, organized in a tree structure that facilitates sorting and searching. Windows manages all available memory as a global heap. Windows' memory manager also supports a local heap in the default data segment of each module in memory.

Instance. In Windows, one of many possible copies of an independently loaded module. Windows can load multiple instances of an application but only one instance of a DLL.

Instance thunk. A short piece of executable code that sets the default data segment for an exported far function before the function executes.

LIM EMS. The Lotus-Intel-Microsoft Expanded Memory Specification describes an industry-standard hardware and software interface to bank-switched (expanded) memory for computers running MS-DOS. Windows' memory manager provides applications with transparent access to expanded memory using version 4.0 of this standard.

Linker. A utility program such as Microsoft's LINK.EXE that combines compiled or assembled object (.OBJ) files and a module-definition (.DEF) file into an application (.EXE) file or a dynamic link library (.DLL) file.

Linking. The process of resolving a program module's external references (references to functions or data in other program modules). Dynamic linking occurs during or after the time a program is loaded into memory to be executed. Static linking, which is performed by a linker such as Microsoft's LINK.EXE, occurs prior to the time a program is loaded.

Loading. The process of copying a module's executable code and data from a file into memory. If the module represents a Windows application, loading allows Windows to begin executing the application. If the module represents a dynamic link library, loading allows the library's functions and data to be accessed by other programs.

Memory model. A description of the layout of executable code and data in memory. Windows programs generally use either a small or a medium memory model. Both small-model and medium-model programs use one data segment; small-model programs use one code segment and medium-model programs can use multiple code segments. With the Microsoft C compiler, you can select a program's memory model by using the */AS* (small-model) or */AM* (medium-model) switch.

Module. A collection of executable code and data that Windows can load into memory. A Windows module must be contained in an executable (.EXE) file or a dynamic link library file, and the name of the file must be the same as the name in the NAME or LIBRARY statement in the module-definition (.DEF) file used to link the module.

Module-definition file. The statements in a module-definition (.DEF) file specify a module's name, type (application or library), segment usage, and imported or exported functions. A linker such as Microsoft's LINK.EXE uses the information in the module-definition file to build segment-loading and dynamic-linking information into a module's loadable .EXE or .DLL file.

Near pointer. A near pointer specifies a memory address as a 16-bit value that represents an offset within a module's default data segment. *See also* Far pointer.

OEM. Original Equipment Manufacturer—that is, the manufacturer of the computer hardware on which Windows software is running. In particular, the "OEM character set" is the character set used in the computer's keyboard and video-display subsystem.

Post. In Windows, to post a message is to place it in an application's message queue, using *PostMessage* or *PostAppMessage*, rather than directly calling an application's window function by using *SendMessage*.

Private data. Data that can be accessed only by a particular function.

Private dynamic link library. A dynamic link library designed to be used by only one application. To designate a DLL as private, use the */P* switch with the resource compiler, RC.EXE.

Protected mode. A CPU addressing mode in which the CPU protects memory by preventing programs from erroneously accessing blocks of memory that belong to other programs. In 16-bit protected mode (supported on the 80286, 80386, and 80486 microprocessors), the CPU can directly address a total of

16 MB of memory, and a memory address consists of a 16-bit selector and a 16-bit offset. In 32-bit protected mode (80386 and 80486), the CPU can directly address a total of 4 GB of memory, and memory addresses are 32-bit values.

Real mode. A CPU addressing mode and a Windows operating mode in which a total of 1 MB of memory is directly addressable. In real mode, a memory address consists of a 16-bit segment and a 16-bit offset. Real mode is supported on all microprocessors in the Intel 8086 family. *See also* Enhanced mode, Standard mode.

Resource. A set of one or more dynamically loadable data items. In Windows, resources are stored as part of executable program (.EXE) files or dynamic link library files. Windows' resource compiler translates programmer-defined resource definitions (.RC file) into a binary format (.RES file) and subsequently adds the binary resources to a specified executable or library file.

Resource-definition file. The statements in a resource-definition (.RC) file describe a module's dynamically loadable resources, including menus, dialog boxes, and icons. A resource compiler compiles the statements in the .RC file to a binary format (.RES file); the binary resources can then be merged into the module's loadable .EXE or library file.

Scaffolding. Source code added to a program specifically to support application development or debugging.

SDK. Software Development Kit.

Send. In Windows, to send a message is to call a window function directly, using *SendMessage*, instead of placing the message in an application's message queue, using *PostMessage* or *PostAppMessage*.

Standard mode. A Windows operating mode in which Windows applications run in 16-bit protected mode on an 80286, 80386, or 80486 microprocessor.

Task. One of several concurrently executing programs. In Windows, tasks are executed cooperatively—that is, each task periodically transfers control to Windows' task manager to let other tasks execute in turn.

Thunk. A short piece of executable code, compiled dynamically by Windows at the time a program is loaded or executed, that performs some simple function on Windows' behalf. *See also* Instance thunk.

Virtual 8086 mode. A CPU addressing mode in which an 80386 or 80486 microprocessor emulates 8086 real-mode addressing. The microprocessor can

concurrently support multiple virtual 8086 sessions. Windows uses virtual 8086 sessions when operating in enhanced mode to execute non-Windows MS-DOS applications.

Virtual memory. A memory-management technique in which data in physical memory can be swapped to disk. The use of virtual memory provides programs with more memory storage and a larger range of memory addresses than could be supported in physical memory alone. Windows' memory manager supports virtual memory when running in enhanced mode on an 80386 or 80486 microprocessor.

Windows application. An executable program designed to run in the Windows environment.

Index

Special Characters

/3 WDEB386 switch, 34
386 enhanced mode, 4, 5, 34, 256
80286 microprocessors, 258–59
80386 microprocessors, 256, 258–59

A

Actor language, 156
AddAtom, 147
Add Custom Control command, 126
/AM compiler switch, 33
Application program interface (API)
  debugging and, 40–43
  functions (*see particular function*)
  modules and, 4, 5
  overview, 3, 255
applications. *See also particular application*
  bulletproofing, 56–58
  DLLs vs., 61, 75
  messages and, 7–8
  multitasking and, 6, 7
  sharing data between (*see* DDE)
  structure of, 13–26
ASCII, Clipboard data format, 171
assembly language, multitasking and, 6
asynchronous events, programming techniques involving, 245–49
atoms, 147, 167–68, 255
automatic data, debugging and, 44–45
automatic variables, debugging and, 45, 46
AuxPrint, 76
AX register, 49, 73

B

bitmaps, 12, 171
breakpoints. *See* debugging
brushes, handles for, 12
bugs. *See* debugging
Button control class, 85, 105

C

C language, 33
C++ language, 156
callback function, 187–96
CallWindowProc, 138, 156
CB_INSERTSTRING, 222
cfFormat, 171
classes, object-oriented programming, 139–43
clients, DDE, 160
Clipboard
  data formats, 171, 200–201
  data transfer and, 159
Close command, 230
CodeView program
  message tracing and, 39
  overview, 31–32
  scaffolding and, 38
  trapping wild pointers and, 54–55
.COD files, 34
/CO linker switch, 32
color, module for, 5
COLOR_BACKGROUND, 44
ColorCtl custom control class, 105–28, 131–32
ColorCtlDlgFn, 126, 128
COLORCTL.DLL, 105–28
ColorCtlFlags, 126, 128
ColorCtlInfo, 126
ColorCtlStyle, 126, 127

ColorCtlWndFn, 126
COLOR_WINDOW, 44
ComboBox control class, 85
commands. *See also particular command*
  in DDE, 164, 198–99, 201
  system, 229–32
Command Status data items, 202
COMM.DRV module, 5
communications, module for, 5
controls
  custom (*see* custom control classes)
  handles for, 11–12
  owner-draw, 221–29
  predefined classes, 85
  programming techniques involving, 207–29
conversations, DDE
  described, 159–61
  initiating in DDEML, 196–98
  managing, 176, 182–83
CONVINFO, 198
cooperative multitasking, 6
CPU modes, bug-proofing and, 56–57
CreateBitmap, 167
CreateCompatibleBitmap, 167
Create50EditWindows, 54
CreateWindow
  in custom controls and, 85, 104, 207
  in debugging, 44, 53–54
  handles and, 12
  in object-oriented programming, 140, 143
CS_DBLCLKS, 126
CS_GLOBALCLASS, 103, 126
cursor
  controlling, 5, 147–48
  handles for, 12
custom control classes. *See also particular class*
  Dialog Editor and
    exit function, 126
    Flags function, 105, 127
custom control classes, Dialog Editor and, *continued*
    Info function, 105, 126
    initialization and class registration, 126
    overview, 104–6
    sample code, 106–25
    Style function, 105, 127, 128
    window function, 126
  in DLLs, 85, 95–104
  examples, 85–95, 96–104
  guidelines for building, 128–32
  painting, 130–32
  testing and debugging, 85
  using, 104
/C: WDEB386 switch, 34

D

data
  clobbered, 48–49
  confusing static and automatic, 44–45
  corruption via wild pointers, 55
  objects and, 143–47
  sharing in DLLs, 77–79
  transfer (*see* DDE)
data formats, DDE and, 171
Data Interchange Format (DIF), 171
data segment(s)
  dissociated, 49–51
  DLLs and, 70–71, 74–76
  register, 49, 73, 75
/D compiler switch, 38
DDE (Dynamic Data Exchange)
  clients and servers, 160
  conversations (*see* conversations, DDE)
  data formats, 171
  design issues, 199–202
  flags, 168–71
  GDI objects, 167
  Management Library (*see* DDEML)
  message-based, 159, 160–64
  OLE vs., 203

DDE (Dynamic Data Exchange), *continued*
overview, 159
service, topic, and item names, 164–66, 199–200
shareable global memory, 166–67, 169
transactions (*see* transactions, DDE)
windows and, 9
DdeAbandonTransaction, 177, 185
DdeAccessData, 179, 187
DdeAddData, 179, 187
DDEADVISE, 166, 168–71
DdeClientTransaction, 177, 183–85, 196, 198
DdeCmpStringHandles, 178, 186
DdeConnect, 176, 182, 196, 198
DdeConnectList, 176, 182, 196, 198
DdeCreateDataHandle, 179, 187
DdeCreateStringHandle, 178, 186
DDEDATA, 166–67, 168–71
DdeDisconnect, 176, 183
DdeDisconnectList, 176, 183
DdeEnableCallback, 177, 185–86
DdeFreeDataHandle, 179, 187
DdeFreeStringHandle, 178, 186
DdeGetData, 179, 187
DdeGetLastError, 175, 182
DdeInitialize, 175, 180, 181, 193
DdeKeepStringHandle, 178, 186
DDEML (DDE Management Library)
API functions
conversation management, 176, 182–83
interface management, 175, 180–82
memory management, 179, 187
string management, 178, 186
transaction management, 177–78, 183–86
callback function, 187–96
establishing data link, 198
executing commands, 198–99
initiating a conversation, 196–98
overview, 159, 172–74
DDEML (DDE Management Library), *continued*
requesting data from a server, 198
sending data to server, 198
transaction processing, 174–75
DdeNameService, 176, 182
DDEPOKE, 166–67, 168–71
DdePostAdvise, 177, 186
DdeQueryConvInfo, 176, 183, 196, 197
DdeQueryNextServer, 176, 183
DdeQueryString, 178, 186
DdeSetUserHandle, 178, 186
DdeUnaccessData, 179, 187
DdeUninitialize, 175, 180, 181
debugging
bulletproofing, 56–58
confusing static and automatic data, 44–45
fatal-exit errors, 45–46
program design and, 29–30
techniques, 37–43
terminals, 30–31
tools, 31–37
visible bugs, 44
wild pointers, 47–56
DEBUG program, 32
default data segment, 70–71, 74–76, 255
#define directive, 38
DefWindowProc
in custom controls, 130
in debugging, 44
non-client areas and, 233, 243–45
in object-oriented programming, 137, 138
system commands and, 231, 232
in window function, 25
descriptor tables, 51
device drivers, modules for, 5
dialog boxes, custom control classes and, 105
DialogBoxParam, 127
Dialog Editor program. *See* custom control classes, Dialog Editor and

DIF (Data Interchange Format), 171
DISCARDABLE attribute, 70
disks, hard, 5
DispatchMessage, 23–24
dissociated data segments, 49–51
dl es WDEB386 command, 34
DLGEDIT.C, 126
DLLBASE.DLL, 61–67
DLLs (dynamic link libraries)
  calling functions, 72–76
  custom controls in, 85, 95–104
  debugging and, 43
  described, 61, 256
  example, 61–68
  managing segments, 68–71
  resources and, 80–81
  sharing functions and data, 77–80
DOS, Windows programming and, 3
DrawFocusRect, 223
DrawRoundFrame, 244
DS register, 49, 73, 75
dumb terminals, 30–31
Dynamic Data Exchange. *See* DDE
dynamic link libraries. *See* DLLs

E

Easter, 67
Edit control, 85, 208–12, 214–21
Ellipse, 131
EMS (expanded memory specification), 3, 33
EnableWindow, 57
encapsulization, 135
EN_CHANGE, 130
EndDialog, 128
enhanced mode, 4, 5, 34, 256
entry point, 256
EnumWindows, 22, 26
errors. *See* debugging
events, asynchronous, 245–49
exiting Windows
  ExitWindows, 249–54
exiting Windows, *continued*
  Windows Exit Procedure (*see WEP*)
expanded memory, 3, 33, 257
exported functions, 5
extended memory, 3, 256

F

far calls, DLL functions and, 72
far function prologs, 72–73
far pointers
  bugs involving, 51–54
  described, 256
  DLLs and, 75–76, 77–80
FatalExit, 31
fatal-exit errors, 45–46
/Fc compiler switch, 34
files
  .COD, 34
  handles for, 11–12
  include, 30
  symbol, 32, 34
FindWindow, 95
FIXED, 54, 77
flags
  DDE and, 168–71, 191
  Flags function, 105, 127–28
fonts, handles for, 11–12
FreeLibrary, 67, 68
functions. *See also particular function or type*
  described, 5
  in DLLs, 72–76, 79–80
  prototypes, 30

G

GDI (graphics device interface)
  DDE and, 167
  described, 256
  handles for, 12
  module for, 5
GetClassInfo, 140, 141, 144–45

GetClassLong, 140
GetClassName, 144
GetClassWord, 140
GetMessage, 8, 23, 24
GetModuleUsage, 68
GetProcAddress, 79
GetProp, 146, 147, 156
GetStyleInfo, 128
GetWindowLong, 40, 140, 142, 143
GetWindowWord, 40, 140, 143
GlobalAddAtom, 147, 168
GlobalAlloc
  in custom controls, 129
  in debugging, 52, 58
  in DDE, 167
  in DLLs, 70, 77
  in object-oriented programming, 144
  overview, 12, 13
global atoms, 147, 167–68
GlobalDeleteAtom, 168
GlobalFindAtom, 168
GlobalGetAtomName, 168
global heap, 13
GlobalLock, 52, 57, 12, 77, 78
global memory
  blocks, 166–67, 169
  handles, 77–80
GlobalReAlloc, 58, 144, 169
GlobalUnlock, 52–53, 57, 78
global variables, 30
GMEM_DDESHARE, 57, 78, 167
GMEM_FIXED, 54
GMEM_MOVEABLE, 54
GMEM_NOT_BANKED, 57, 78
graphics
  DDE and, 167
  handles and, 12
  module for, 5
  owner-draw controls, 221–29
graphics device interface. *See* GDI
/Gw compiler switch, 49

H

handles
  described, 257
  functions involving, 178–79, 186–87
  global memory, 77–80
  items using, 11–12
  for metafiles, 12, 171
hard disks, 5
hash tables, 147, 257
hBrush variable, 45
hData variable, 46
heaps, memory, 13, 257
Heap Walker utility, 55, 58
HideWaitCursor, 147–48

I

icons
  handles for, 12
  module for managing, 5
import libraries, 43
include files, 30
InflateRect, 131
Info, 105, 126–27
Init, 22
input. *See also* keyboard(s); mouse
  bulletproofing and, 57
  custom control classes and, 129
InstallKeyTrap, 156
instances
  described, 5–6, 257
  handles for, 11–12
  Windows libraries and, 70
instance thunks, 50–51, 73, 257
intercept functions, 40–43
interface management, DDEML, 175,
    180–82
interrupt handlers, 245
item names, 164–66, 199–200

J

jump tables, in DDEML callback
    functions, 192–93

K

KERNEL.EXE module, 5
keyboard(s)
  bulletproofing and, 57
  messages and, 7–8
  module for, 5
KEYTRAP.EXE, 149–56
KeyTrapWndFn, 156
KRNL286.EXE module, 5
KRNL386.EXE module, 5

L

languages, programming. *See particular language*
LAN Technical Reference manual, 246
LB_ADDSTRING, 222, 223
LB_INSERTSTRING, 222
LibEntry, 66, 70
LibMain
  in custom controls, 96, 103, 126
  described, 61, 66
  in memory models, 70
libraries
  custom control, 105–28
  DDE Management (*see* DDEML)
  dynamic link (*see* DLLs)
  import, 43
library reference counts, 68
/LI linker switch, 33
LIM (Lotus-Intel-Microsoft) EMS standard, 3
linker, 257
Lisp language, 145
list-box controls, 85, 207–8, 223–29
ln command, 35
LoadBitmap, 80
LoadCursor, 147
LoadIcon, 80
LoadLibrary
  in custom controls, 104
  described, 4
  in DLLs, 67, 68, 80
  in filtering messages, 214
LoadModule, 4
LoadString, 67, 80
LocalAlloc, 13, 75, 129, 144
local atoms, 147
LocalFree, 37
local heap
  clobbered data and, 48
  described, 13
  DLLs and, 66
LocalInit, 66
LocalLock, 37
LocalUnlock, 37
LockData, 71, 77
LockSegment, 71
LockSegment(-1), 77
Lotus-Intel-Microsoft Expanded Memory Specification standard, 3
low memory, 57–58
lpfnIdToStr, 127
lpfnStrToId, 127

M

MAKEINTATOM macro, 147
MakeProcInstance, 50, 51, 73
/MAP linker switch, 33
MAPSYM utility, 32, 33, 36
Maximize command, 230
memory
  custom control classes and, 129
  expanded, 3, 33
  extended, 3
  global blocks, 166–67, 169
  handles for, 11–12
  low, 57–58
  managing
    DDEML functions, 179, 187
    module for, 5
    utility for, 51–54, 69–70
    Windows and, 3, 12–13
  models, 69–70, 258
  virtual, 3
  wild pointers and, 47–56
  window extra bytes, 143–45

memory manager, 51–54, 69–70
menus, handles for, 11–12
message-based DDE, 159, 160–64
MessageBox, 51
messages
  DDE and, 159, 160–64
  filtering, 149–56, 213–21, 231–32
  object-oriented programming and, 136–37, 138–39
  overview, 6, 7–9
  in sample Windows application, 23–24
  tracing, 39
metafiles, handles for, 12, 171
Microsoft C Compiler, 32, 33–34
Microsoft C Professional Development System, *xv*
Microsoft Excel, 199, 201
Microsoft Linker, 32, 33
Microsoft Object Linking and Embedding protocol, 203
Microsoft Symbolic Link format, 171
Microsoft Windows
  API (*see* Application program interface)
  applications (*see* applications)
  custom control classes and, 85, 103
  debuggers
    CodeView (*see* CodeView program)
    SYMDEB, 29, 32–33, 56
    WDEB386, 29, 33–37
  exiting (*see* exiting Windows)
  functions, 2, 5, 24
  fundamental services, 3–4
  handles, 11–12
  memory and, 3
  memory manager, 51–54, 69–70
  message-passing in, 138–39
  modules, 4, 5
  Software Development Kit (*see* Microsoft Windows Software Development Kit)
  task manager, 3, 5–6, 7
Microsoft Windows Software Development Kit
  debugging tools, 31–37
  Dialog Editor (*see* custom control classes, Dialog Editor and)
  messages and, 7
Minimize command, 230
modes, 4, 5, 34
MODSTAT.EXE, 13–26
modules, 4, 5, 11–12, 258
monitors, 30–31
mouse
  messages and, 7–8
  module for, 5
MOVEABLE, 54, 77
moving windows, 208, 230
MS-DOS, Windows programming and, 3
MsgCommand, 25
MsgEraseBkgnd, 130
/m SYMDEB switch, 32, 33
multitasking, 3, 5–6, 7
MyPrintf, 67–68

N

_NCBPost, 248–49
near pointers, DLLs and, 75–76
NetBIOSCall, 246
NetBIOS protocol, 245–49
networks, 245–49
NODATA, 70
non-client areas, 233–45
Notepad application, 213
NumEditWndFn, 138, 139, 141–42

O

Object Linking and Embedding (OLE) protocol, 203
object-oriented programming
  associating objects and data, 143–47
  classes, 139–43

object-oriented programming, *continued*
  described, 135–36
  message passing, 136–37, 138–39
  sample programs, 147–48, 149–56
  windows as objects, 137–38
  Windows limitations, 135
ODA_DRAWENTIRE, 222
ODA_FOCUS, 222
ODA_SELECT, 222
/Od compiler switch, 32, 33
ODLB.EXE, 223–29
ODS_CHECKED, 222
ODS_DISABLED, 222
ODS_FOCUS, 222
ODS_GRAYED, 222
ODS_SELECTED, 222
OEM character set, 171
OLE (Object Linking and Embedding) protocol, 203
OpenFile, 12
operating environment, procedural vs. message-driven, 136
OS/2, 6, 221
OutputDebugString, 31, 38, 76
output devices, handles for, 12
OVEDIT.EXE, 208–12
owner-draw controls, 221–29

P

palettes, handles for, 12
parameters
  messages and, 7–8
  passing conventions and DLLs, 74
PatBlt, 223
PeekMessage, 8, 24
pens, handles for, 12
Petzold, Charles, *xii*
pointer-arithmetic errors, 48
pointers
  DLLs and, 75–76, 77–80
  far (*see* far pointers)
  wild, 47–56
popup windows, 10, 11
PostAppMessage, 248, 249, 258
PostMessage, 162, 168, 248, 249, 258
PostNCBMessage, 248
PostQuitMessage, 25
preemptive multitasking, 6
PRELOAD, 70
printers, handles for, 12
procedural operating environment, message-driven environment vs., 136
programming, Windows
  books on, *xii, xiv*
  MS-DOS and, 3
  object-oriented (*see* object-oriented programming)
  techniques, 207–54 (*see also particular subject*)
  tools, *xiv*
  using components in, 13
Programming Windows, *xii, xiv*
programs. *See also* applications
  designing, 199–202
  managing segments, 68–71
  source code for book's examples, *xv*
property lists
  atoms and, 147
  overview, 145–46
  sample programs using, 147–48, 149–56
protected mode, 258

Q

quitting Windows. *See* exiting Windows

R

real mode, 4, 5, 259
Rectangle, 131
reference counts, library, 68
regions, handles for, 12

RegisterClass
  in custom controls, 85, 103
  in debugging, 44, 51
  in object classes, 140, 141, 143
  in Windows application structure, 23
RegisterClipboardFormat, 201
RemoveProp, 146, 147
resizing windows, 230
resource editors, custom control classes and. *See* custom control classes, Dialog Editor and
resources
  defining within DLLs, 80–81
  described, 259
  handles for, 11–12
Restore command, 230
ROEDIT.DLL, 214–20
ROEditWindFn, 220–21
rounded windows, 233–45
ROUND.EXE, 233–43
RoundRect, 131
RoundWndFn, 233, 243–45
RYG control class, 85–95, 96–104
RYGDEV.EXE, 86–94
RYG.DLL, 96–104

S

scaffolding (debugging technique), 37–38, 259
SC_CLOSE, 230
SC_HSCROLL, 230
SC_KEYMENU, 230
SC_MAXIMIZE, 230
SC_MINIMIZE, 230
SC_MOUSEMENU, 230
SC_MOVE, 230, 232
SC_NEXTWINDOW, 230
SC_RESTORE, 230
scroll bars, controlling, 85, 230
SC_SIZE, 230, 232
SC_TASKLIST, 230
SC_VSCROLL, 230
SDK. *See* Microsoft Windows Software Development Kit
SendMessage, 162
servers, DDE, 160
service names, 164–66, 199–200
SetClassLong, 140
SetClassWord, 140
SetCursor, 147, 244
SetProp, 145–46, 147
SetWindowLong, 40, 49, 140, 142, 143
SetWindowWord, 40, 49, 140, 143
Shaker utility, 55
ShowDLLIcon, 67
ShowModuleInfo, 25–26
ShowWaitCursor, 147–48
ShowWindow, 44
sizing windows, 230
Smalltalk-80 language, 135, 138
Software Arts' Data Interchange Format, 171
Software Development Kit. *See* Microsoft Windows Software Development Kit
sound, module for, 5
source code
  for book's program examples, *xv*
  debugging and, 29–30
Spy utility, 39
SS register, 75
stack, DLLs and, 74–76
stack-frame pointer, 48
stack segment register, 75
standard mode, 4, 5, 259
Static control class, 85, 105
static data
  debugging and, 44–45, 50
  DLLs and, 70
static variables, 30
status flags, DDE and, 170–71
strings
  DDEML management functions, 178, 186
  handles for, 12

STRINGTABLE resource, 67
Style, 105, 127, 128
subclasses, 140–42
/s WDEB386 switch, 34
/S: WDEB386 switch, 34
switch statements
  DDEML callback function and, 190
  described, 25
  object structure and, 135
  tracing messages and, 39
symbol files, 32, 34
Symbolic Link format, 171
SYMDEB program, 29, 32–33, 56
system commands, programming techniques involving, 229–32
SYSTEM.DRV module, 5
system timer, module for, 5
System topic, 200–201

T

Tagged Image File Format. *See* TIFF
task manager, 5–6, 7
terminals, debugging, 30–31, 32
TextOut, 25
thunks, instance, 50–51, 73, 257, 259
TIFF (Tagged Image File Format), 171
topic names, 164–66, 199–200
TopLevelWndFn, 22, 25, 208
transactions, DDE
  described, 159–60
  managing, 177–78, 183–86
  processing, 174–75
TranslateMessage, 23

U

uninitialized pointer variables, 47–48
UninstallKeyTrap, 156
UnlockData, 66, 71
UnlockSegment, 66, 71
UnregisterClass, 95, 104
UpdateWindow, 44
USER.EXE module, 5

V

ValidateCodeSegments, 55–56
ValidateFreeSpaces, 55–56
variables
  automatic, 45, 46
  debugging and, 30
  global, 30
  uninitialized pointer, 47–48
video
  custom control classes and, 130–32
  messages and, 8
  objects and, 143
virtual 8086 mode, 259
virtual memory, 3, 260

W

WaitMessage function, 8, 24
WC_NCCALCSIZE, 243
WC_NCPAINT, 244
WDEB386 program, 29, 33–37
WEP (Windows Exit Procedure)
  custom controls and, 96, 103–4, 126
  overview, 61, 67–68
wHeapSize, 66
wild pointers, 47–56
window extra bytes, 143–45
Windows. *See* Microsoft Windows
windows
  bugs affecting appearance, 44
  closing, 230
  extra bytes, 143–45
  moving, 230
  non-client areas, 233–45
  as objects, 137–38
  overview, 9–11
  rounded, 233–45
  sizing, 230
Windows Exit Procedure. *See WEP*

Windows Software Development Kit. *See* Microsoft Windows Software Development Kit
WinExec, 4, 95
WINEXIT.EXE, 249–54
WinMain, 22, 23, 61, 104
wMaxLen, 128
WM_CHAR, 138
WM_COMMAND, 129–30
WM_CREATE, 45
WM_CTLCOLOR, 45, 130
WM_DDE_ACK, 160–62, 164, 168–71
WM_DDE_ADVISE, 160, 161, 163, 164
WM_DDE_DATA, 160–64, 166, 168–69
WM_DDE_EXECUTE, 161, 163, 201–2
WM_DDE_INITIATE, 160–62, 163, 168
WM_DDE_POKE, 161, 163, 164, 166
WM_DDE_REQUEST, 160, 161, 163, 164
WM_DDE_TERMINATE, 160, 161, 166
WM_DDE_UNADVISE, 160, 161, 163, 164
WM_DRAWITEM, 130, 221–22, 223
WM_ERASEBKGND, 130, 244
WM_KEYDOWN, 136, 156
WM_LBUTTONDOWN, 232
WM_MOUSEMOVE, 136
WM_NCCALCSIZE, 131
WM_NCPAINT, 131, 233
WM_PAINT, 131, 136
WM_SETCURSOR, 232, 233, 244
WM_SETFOCUS, 138
WM_SETFONT, 130
WM_SYSCOMMAND, 229–32, 233, 244
WNDCLASS, 44, 49, 140, 143
WndEnumFn, 22, 26
WndFn, 45
WS_BORDER, 130–31
WS_CAPTION, 207
WS_CLIPSIBLINGS, 244–45
WS_THICKFRAME, 207
WS_VISIBLE, 44
wvsprintf, 76

X

XCLASS_BOOL, 191
XCLASS_DATA, 191
XCLASS_FLAGS, 191
XCLASS_NOTIFICATION, 191
xGWL, 40
xGWW, 40
xSWL, 40
xSWW, 40
XTYP_ADVDATA, 188, 196
XTYP_ADVREQ, 188, 195
XTYP_ADVSTART, 188, 195, 198
XTYP_ADVSTOP, 188, 195
XTYP_CONNECT, 189, 194
XTYP_CONNECT_CONFIRM, 189, 194
XTYP_DISCONNECT, 190, 194
XTYP_EXECUTE, 188, 195, 198–99, 201–2
XTYPF_NOBLOCK, 191
XTYP_MONITOR, 190
XTYP_POKE, 189, 195, 198
XTYP_REGISTER, 189, 194
XTYP_REQUEST, 190, 195, 198
XTYP_UNREGISTER, 190, 194
XTYP_WILD_CONNECT, 190, 194
XTYP_XACT_COMPLETE, 189, 196, 197, 198–99

Y

y 386env WDEB386 command, 34

Z

/Zd compiler switch, 33
/Zi compiler switch, 32
z WDEB386 command, 34

Richard Wilton

Richard Wilton has been programming computers since the late 1960s. He has written systems software and graphics applications in FORTRAN, Pascal, C, Forth, and assembly language. His articles and reviews have appeared in several computer publications, including *BYTE* and *Computer Language*. Currently an assistant professor of pediatrics at the University of California, Los Angeles, he earned an M.D. from UCLA and completed his residency in pediatrics at the Childrens Hospital of Los Angeles. He uses Windows and DDE in a patient-tracking database system in the pediatrics clinics at UCLA. He is the author of *Programmer's Guide to PC & PS/2 Video Systems* and is coauthor of *The New Peter Norton Programmer's Guide to the IBM PC & PS/2*, both published by Microsoft Press.

The manuscript for this book was prepared and submitted to Microsoft Press in electronic form. Text files were processed and formatted using Microsoft Word.

Principal word processors: Rodney Cook and Katherine Erickson

Principal proofreader: Cynthia Riskin

Principal typographer: Ruth Pettis

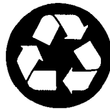
Interior text designer: Kim Eggleston

Principal illustrator: Lisa Sandburg

Cover designer: Tom Draper

Cover color separator: Color Service, Inc.

Text composition by Microsoft Press in Garamond Light with display type in Futura Extra Bold, using the Magna composition system and the Linotronic 300 laser imagesetter.



Printed on recycled paper stock.

SPECIAL OFFER

Companion Disk for MICROSOFT® WINDOWS™ 3 DEVELOPER'S WORKSHOP

Microsoft Press has created a timesaving Companion Disk for MICROSOFT WINDOWS 3 DEVELOPER'S WORKSHOP in both 5 1/4-inch format (one 1.2MB disk) and 3 1/2-inch format (one 1.44 MB disk) that contains

- source code for all the programs in the book
- complete programs that use the source-code fragments given as examples in the book
- two programs—a server and client application—that fully demonstrate the DDE management library
- several bonus applications (not in the book) that you can examine to hone your Windows programming skills

The companion disk for MICROSOFT WINDOWS 3 DEVELOPER'S WORKSHOP—available only from Microsoft Press—is a valuable, ready-to-use resource for Windows programmers. Order your disk today!

Domestic Ordering Information:
To order, use the special reply card in the back of the book. If the card has already been used, please send $24.95, plus sales tax in the following states if applicable: AZ, CA, CO, CT, DC, FL, GA, HI, ID, IL, IN, IA, KS, KY, ME, MD, MA, MI, MN, MO, NE, NV, NJ, NM, NY, NC, OH, OK, PA, RI, SC, TN, TX, VA, WA, WV, WI. Microsoft reserves the right to correct tax rates and/or collect the sales tax assessed by additional states as required by law, without notice. Please add $2.50 per disk set for domestic postage and handling charges. Mail your order to: **Microsoft Press, Attn: Companion Disk Offer, 21919 20th Ave SE, Box 3011, Bothell, WA 98041-3011.** Specify 5 1/4-inch or 3 1/2-inch format. Payment must be in U.S. funds. You may pay by check or money order (payable to Microsoft Press) or by American Express, VISA, or MasterCard; please include credit card number, expiration date, and cardholder signature. Allow 2–3 weeks for delivery.

Foreign Ordering Information (except within the U.K. and Canada., see below):
Follow procedures for domestic ordering. Add $15.00 per disk set for foreign postage and handling.

U.K. Ordering Information:
Send your order in writing along with £22.95 (includes VAT) to: Microsoft Press, 27 Wrights Lane, London W8 5TZ. You may pay by check or money order (payable to Microsoft Press) or by American Express, VISA, MasterCard, or Diners Club; please include credit card number, expiration date, and cardholder signature. Specify 5 1/4-inch or 3 1/2-inch format.

Canadian Ordering Information:
Send your order in writing along with $32.95 (includes GST) to: Macmillan Canada, Attn: Microsoft Press Department, 164 Commander Blvd., Agincourt, Ontario, Canada M1S 3C7. You may pay by check or money order (payable to Microsoft Press) or by VISA or MasterCard; please include credit card number, expiration date, and cardholder signature. Specify 5 1/4-inch or 3 1/2-inch format.

Microsoft Press Companion Disk Guarantee:
If a disk is defective, a replacement disk will be sent. Please send the defective disk along with your packing slip (or copy) to: Microsoft Press, Consumer Sales, One Microsoft Way, Redmond, WA 98052-6399. Send your questions or comments about the files on the disk to: Win 3 Developer's Disk, Microsoft Press, One Microsoft Way, Redmond, WA 98052-6399.

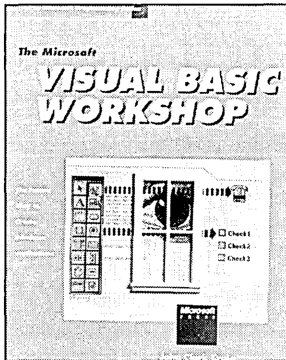
Information on Obtaining DDEML.DLL

Chapter 6 of this book contains information about Microsoft's Dynamic Data Exchange (DDE) Management Library. To obtain a copy of the DDEML.DLL library, which you will need in order to use the examples in Chapter 6, write to Microsoft Press, Attn: DDEML.DLL, One Microsoft Way, Redmond, WA 98052-6399.

No phone calls please.

NOTE: DDEML.DLL is a part of the Windows 3.1 beta release. If you or your company is a Windows 3.1 beta site, you already have DDEML.DLL and do not need to write to the address above.

In-depth Windows™ 3 Programming Resources from Microsoft Press



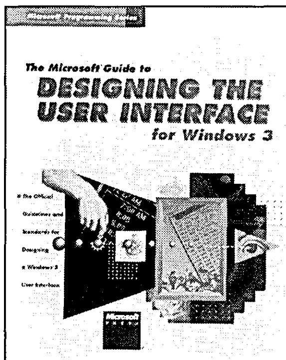
THE MICROSOFT® VISUAL BASIC™ WORKSHOP

John Clark Craig

Create Windows applications quickly with Microsoft Visual Basic and THE MICROSOFT VISUAL BASIC WORKSHOP. Whether you're new to Windows programming or are a seasoned programmer, you'll find this book-and-software package invaluable. It includes dozens of ready-to-use Visual Basic routines and applications that can be easily incorporated into your Windows programming projects. The author provides helpful overviews of both Visual Basic and event-driven programming. Also covered are advanced programming concepts—using Dynamic Data Exchange (DDE), using the Windows Applications Programming Interface (API), creating Dynamic Link Libraries (DLL's), and using Windows graphics.

NOTE: Both executable and source-code files are included so you can preview Visual Basic if you don't already own it!

420 pages, softcover with one $5\frac{1}{4}$ 1.2MB disk $7\frac{3}{8} \times 9\frac{1}{4}$ $39.95
Order Code VIBAWO



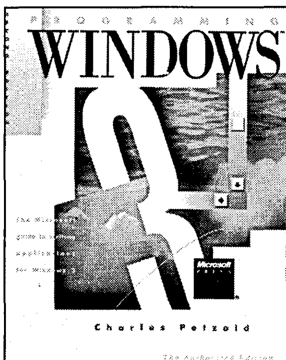
THE MICROSOFT® GUIDE TO DESIGNING THE USER INTERFACE

For Windows™ 3 Applications

Microsoft Corporation

This is the official guide to the standards for creating a well-designed and functionally consistent Windows 3 interface. General design principles as well as specifics on individual graphical elements—windows, scroll bars, icons, and dialog boxes—are explored and discussed. In addition, there is detailed information on the standard methods for providing user with the mechanisms for customizing the user interface and adding a help system.

350 pages, softcover $7\frac{3}{8} \times 9\frac{1}{4}$ $27.95 Order Code GUDEUS
Available December 1991



PROGRAMMING WINDOWS,™ 2nd ed.

The Microsoft® Guide to Writing Applications for Windows™ 3

Charles Petzold

"PROGRAMMING WINDOWS by Charles Petzold is an excellent reference. This remains the classic Windows programming guide." **Programmer's Journal**

This new edition of PROGRAMMING WINDOWS—completely updated and revised to highlight version 3 capabilities—is once again packed with keen insight, tried-and-true programming techniques, scores of complete sample programs written in C, and straightforward explanations of the Microsoft Windows programing environment. New chapters detail Dynamic Data Exchange (DDE) and the Multiple Document Interface (MDI) features. Other topics include: reading input, using resources, the graphics device interface (GDI), and data exchange and links.

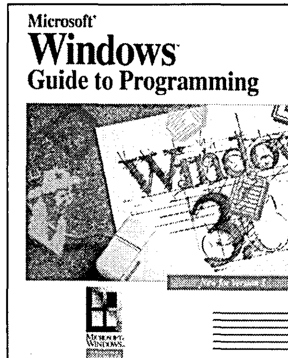
960 pages, softcover $7\frac{3}{8} \times 9\frac{1}{4}$ $29.95 Order Code PRWI2

Microsoft Windows Programmer's Reference Library

The core documentation—including both technical data and programming tutorials—that Microsoft provides with the Microsoft Windows Software Developer's Kit (SDK) can now be purchased separately. These three volumes provide the most accurate and up-to-date Windows 3 programming information available.

"If you intend to do any serious Windows programming, these books are a must. They provide virtually everything you may want to know about how to program in C for and in the Windows environment."

PC Techniques

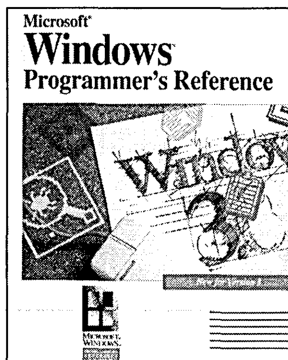


MICROSOFT® WINDOWS™ GUIDE TO PROGRAMMING

Microsoft Corporation

An example-packed introduction to writing applications using the Microsoft Windows version 3 application programming interface (API). Specifically written for the C programmer who wants to learn how to use Windows' functions, messages, and data structures to build efficient and reliable applications. Step-by-step instruction accompanied by dozens of sample applications that can be compiled and run with Windows.

560 pages, softcover $7\frac{3}{8}$ x $9\frac{1}{4}$ $29.95 Order Code WIGUPR

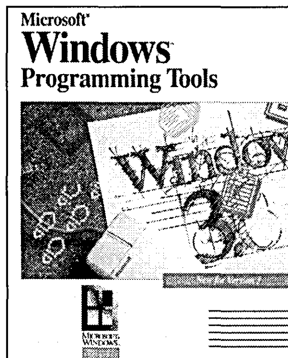


MICROSOFT® WINDOWS™ PROGRAMMER'S REFERENCE

Microsoft Corporation

An up-to-date, comprehensive reference to each component in the Windows 3 application programming interface (API). Indispensable to every Windows programmer, this information is the foundation for any program that takes advantage of Windows' special capabilities.

1152 pages, softcover $7\frac{3}{8}$ x $9\frac{1}{4}$ $39.95 Order Code WIPRRE



MICROSOFT® WINDOWS™ PROGRAMMING TOOLS

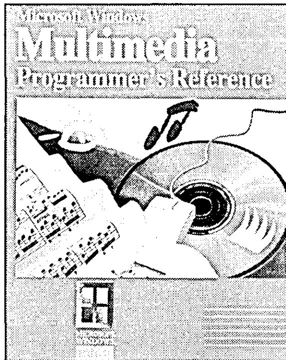
Microsoft Corporation

Detailed instruction on using the programming tools that come with the Microsoft Windows SDK. This book examines how the Help system elements combine to produce a system helpful to the user, and it explains in detail how to plan, write, and compile a working Windows Help system.

400 pages, softcover $7\frac{3}{8}$ x $9\frac{1}{4}$ $24.95 Order Code WIPRTO

Microsoft® Windows™ Multimedia Programmer's Reference Library

The Multimedia extensions to Microsoft Windows offer an outstanding opportunity for developers who want to add a high level of music, audio, animation and other multimedia elements to their programs. These three volumes on Microsoft Windows with Multimedia extensions are the official documentation to the Microsoft Multimedia Development Kit (MDK). They provide the most up-to-date and accurate information available and are a great way to preview the MDK.

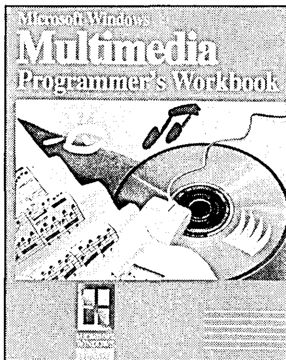


MICROSOFT® WINDOWS™ MULTIMEDIA PROGRAMMER'S REFERENCE

Microsoft Corporation

The essential reference for application programmers working with Microsoft Windows with Multimedia. Includes complete coverage of the Application Programming Interface (API), its messages, data types, structures, and file formats.

432 pages, softcover 7 3/8 x 9 1/4 $27.95 Order Code MMPRRE

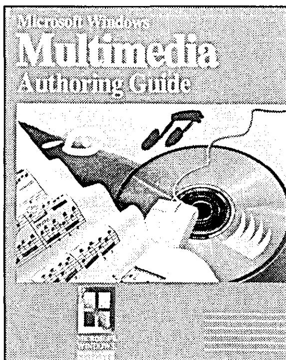


MICROSOFT® WINDOWS™ MULTIMEDIA PROGRAMMER'S WORKBOOK

Microsoft Corporation

This companion volume to the Programmer's Reference provides an overview of the Microsoft Windows with Multimedia architecture and the Media Control Interface (MCI). In addition, instruction and sample C code demonstrate how to use all the multimedia audio, video, and hardware device functions.

352 pages, softcover 7 3/8 x 9 1/4 $22.95 Order Code MMPRWO



MICROSOFT® WINDOWS™ MULTIMEDIA AUTHORING AND TOOLS GUIDE

Microsoft Corporation

A very accessible look at the high-level multimedia development tools and processes that are involved in integrating image, sound, and text data within a multimedia title. It is a fact-filled hardware and software planning guide—ideal for anyone who wants to create titles for a multimedia PC. In addition, the book surveys the data preparation tools that are included with the MDK.

320 pages, softcover 7 3/8 x 9 1/4 $24.95 Order code MMAUGU

Microsoft Press books are available wherever quality computer books are sold.
Or call **1-800-MSPRESS** for ordering information or placing credit card orders.
Please refer to **BBK** when placing your order.

In Canada, contact Macmillan of Canada, Attn: Microsoft Press Dept., 164 Commander Blvd., Agincourt, Ontario, Canada M1S 3C7. 416-293-8141

In the U.K., contact Microsft Press, 27 Wrights Lane, London W8 5TZ.

Microsoft University

A New Perspective on Windows Training

Microsoft University courses take you to the heart of our microcomputer software architecture. Lab sessions provide practical, hands-on experience and show you how to develop and debug software more efficiently. Our qualified instructors explain the philosophy and principles that drive our system design.

Windows Courses for Support Personnel

Our courses train professional support engineers to help end users overcome the operational difficulties they may encounter with Microsoft Windows.™ Students receive hands-on experience installing Microsoft Windows and configuring the system. We also provide support courses which help to maximize system efficiency. The *MS-DOS 5 Installation and Optimization* video course consolidates all the information you need to successfully install, optimize, customize and support MS-DOS® 5 on a group of PCs. In the *Microsoft Windows for Support Engineers* course, students learn how to manage memory usage of a Windows session, optimize PC system resources for Windows, troubleshoot common problems encountered by end users, and customize a Windows environment and the installation process.

Windows Courses for Developers

We offer courses for experienced software developers who are new to leveloping applications for the Microsoft Windows graphical environment, s well as courses for more seasoned Windows developers. For example, we ffer the *Microsoft Visual Basic Programming* course which teaches the :atures and capabilities of the Visual Basic™ programming system, as well the concepts needed to write sophisticated, event-driven graphical programs. The *Fundamentals of Microsoft Windows Programming,* an intensive, hands-on video course, prepares students for more advanced Windows courses. *Exploring Controls* is a definitive video course on Microsoft Windows controls, including buttons, static controls, edit controls, list boxes, combo boxes, scroll bars, and custom controls.

Custom Training Options

Microsoft University will deliver any of its courses at your location, and we can accommodate as many as 20 students per class, which significantly lowers the cost per student. Companies may also license the Microsoft Windows for Support Engineers course and have their own internal training organizations deliver the course. Microsoft University will also customize any of its Windows courses to meet your unique training needs, and we are happy to provide an estimate for this service.

Take Microsoft University Windows Classes Closer to Home

We understand the pressures of the marketplace and know what a difference timely training can make in the success of your Windows project. That's why we offer regularly scheduled Windows classes at our regional facilities: Seattle, Boston, San Francisco, Los Angeles, New York, Chicago, Atlanta, Dallas, Washington, D.C., and Toronto. In addition, we have authorized training centers across the country which now offer Windows courses for support personnel.

To receive our current Course Schedule, which provides course outlines and complete registration information, please call Microsoft University at (206) 828-1507, and ask for department 611.

I'D LIKE TO KNOW MORE!

☐ Please send me the most current Microsoft University course schedule.

☐ Please have a representative call me regarding hosting a Microsoft University course at our facility.

Course / Topic

Please indicate your area of interest:

☐ Microsoft C
☐ Microsoft SQL Server
☐ Microsoft Windows
☐ Microsoft® OS/2®
☐ Microsoft LAN Manager
☐ Microsoft OS/2 Presentation Manager
☐ Microsoft Visual Basic™
☐ MS-DOS® 5

☐ Please send me information on custom courses.

Course / Topic

code 611

PLEASE PRINT

Name

Job Title/Position

Company

Street Address

City State Zip Code

Daytime Phone

Please clip along dotted line and mail to:

Microsoft University

MSU

10700 Northup Way Bellevue, WA 98004-1447

Microsoft, the Microsoft logo, MS, and MS-DOS are registered trademarks, and Windows and Visual Basic are trademarks of Microsoft Corporation. OS/2 is a registered trademark. Presentation Manager is a trademark licensed to Microsoft Corporation.

Microsoft®

Microsoft®

WINDOWS™ 3

Developer's Workshop

"After you've nailed down the basics, Rick Wilton's valuable book will surely sharpen your Windows programming skills." Charles Petzold

An example-packed resource for every Microsoft Windows programmer.

Richard Wilton provides richly detailed discussions of some of the central—and most complex—issues that every Windows programmer tackles. Wilton's approach is practical, with a clear focus on problem solving. He demonstrates how to blend good programming practices with a well-conceived design. Scores of source-code examples illustrate and amplify the discussions. Topics include:

Debugging. Designing for debugging; using the built-in Windows debugging tools; avoiding the most common bugs; tracking down wild pointers; "bulletproofing" your code.

Dynamic Link Libraries (DLL). Sharing executable code and data with DLLs; structuring DLLs to use memory effectively; calling library functions.

Custom Controls. Implementing a custom control class in a DLL; making custom controls available to resource editors; building the perfect custom control.

Windows and Objects. Understanding the object-oriented roots of Windows; using object-oriented concepts to design better source code.

Dynamic Data Exchange (DDE). Implementing direct interprocess communication with DDE; designing a DDE application using low-level message passing and memory management; understanding and using the added functionality of the DDE programming interface, the DDE Management Library (DDEML).

In addition, Wilton offers practical approaches to programming puzzles that include wresting additional functionality from predefined control classes, customizing the nonclient area, and handling asynchronous events.

U.S.A.	**$24.95**
U.K.	£32.95
Canada	$22.95

[Recommended]

The Authorized Editions

ISBN 1-55615-244-2

Programming / Windows