

Designed for
Microsoft®
Windows® 95



CD-ROM
Included

Microsoft
Programming
Series

Programming

Windows® 95



The Definitive
Developer's
Guide to the
Windows 95 API

Charles Petzold

Microsoft Press



Programming

Windows® 95

The Classic Guide to Writing Applications for Windows 95

Updated for the 32-bit world of Windows 95, this Charles Petzold bestseller is the best-known, most widely praised, and most widely used how-to programming book on the planet. For a generation of C programmers, "look it up in Petzold" has been the final word on questions about programming for Windows. In this thorough revision, Petzold reveals many valuable new insights, augmented by key contributions from programming expert Paul Yao. Still the comprehensive reference and tutorial to the core areas of programming for Windows, PROGRAMMING WINDOWS 95 is now a 32-bit book with 32-bit programs on CD-ROM, covering new Windows 95 topics such as:

- ◆ Multithreading
- ◆ GDI and OLE enhancements
- ◆ Preemptive multitasking
- ◆ Printing and memory (both completely revamped)
- ◆ The new user interface

Now extensively revised, PROGRAMMING WINDOWS 95 is the most authoritative, example-packed, and thorough resource for programmers writing applications for the Microsoft® Windows 95 operating system. No aspiring or experienced developer can afford to be without it.

Charles Petzold is a distinguished freelance writer well known for his books and articles on Windows and OS/2®. He was honored in 1994 with the Windows Pioneer Award, presented by Bill Gates and *Windows Magazine*, for his unique contribution to programming for Windows. The other "Windows Pioneers" were Alan Cooper, Lyle Griffin, Joe Guthridge, Ted Johnson, Ian Koenig, and Ray Ozzie.

Paul Yao is the president of The Paul Yao Company, a Seattle-based firm that has specialized in Windows programmer training since 1986. Paul teaches workshops on native OLE, device drivers, Win32®, Win16, and MFC worldwide. This is the fifth book he has coauthored on programming for Windows.

Charles Petzold
updated with Paul Yao



CD-ROM Included! Valuable Information Included on CD!

The companion CD-ROM contains source code and EXE files for all the book's programs, including many useful utilities that let you explore and understand Windows' use of:

- ◆ Keyboard
- ◆ Mouse
- ◆ Memory
- ◆ Graphics
- ◆ Color
- ◆ Fonts

System Requirements:

Microsoft Windows 95 operating system and Microsoft® Visual C++® version 4.0

► **"If you're going to program for Windows, buy this book. It will pay for itself in a matter of hours."**

Computer Language

► **"A thorough introduction for anyone who wants to look behind the curtain."**

050617 wis,
TI nes

4245

Operating Systems/Windows 95/C Programming



7 90145 56766 6

U.S.A. \$49.95
U.K. £46.99 [V.A.T. included]
Canada \$67.95
[Recommended]

Microsoft Press

ISBN 1-55615-676-6
9781556156762
X0023S5UFX
Programming Windows 95 (Mic...
USED CP G
Box # 185

9 "781556"156762"

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

Copyright © 1996 by Charles Petzold

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
Petzold, Charles, 1953--
Programming Windows 95 / Charles Petzold. -- 4th ed.

p. cm.

Includes index.

ISBN 1-55615-676-6

1. Microsoft Windows (Computer file) 2. Operating systems
(Computers) I. Title.

QA76.76.O63P533 1996

005.265--dc20

95-49555
CIP

Printed and bound in the United States of America.

2 3 4 5 6 7 8 9 QMQM 1 0 9 8 7 6

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

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (206) 936-7529.

PageMaker is a trademark of Adobe Systems, Inc. Apple, LaserWriter, Lisa, Macintosh, and TrueType are registered trademarks of Apple Computer, Inc. ToolBook is a registered trademark of Asymetrix Corporation. Borland, Delphi, SideKick, and Turbo C are registered trademarks of Borland International, Inc. Corel and Design is a registered trademark of Corel Systems Corporation. DEC is a trademark of Digital Equipment Corporation. Digital Research and GEM are registered trademarks of Digital Research, Inc. ColorPro, Hewlett-Packard, and HP are registered trademarks of Hewlett-Packard Company. IBM, OS/2, and TopView are registered trademarks and Corel and Graphics Assistant are trademarks of International Business Machines Corporation. Lotus and VisiCalc are registered trademarks of Lotus Development Corporation. Micrografx Designer is a trademark of Micrografx, Inc. Microsoft, Microsoft Press, Microsoft Press and Design, MS-DOS, MultiPlan, QuickC, Visual Basic, Windows, and the Windows operating system logo are registered trademarks of Microsoft Corporation. NEC is a registered trademark of NEC Corporation. PowerBuilder is a trademark of Powersoft Corporation. DESQview is a registered trademark of Quarterdeck Office Systems. Visio is a registered trademark of Visio Corporation. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. Xerox is a registered trademark of Xerox Corp. All other trademarks and service marks are the property of their respective owners.

Acquisitions Editor: Eric Stroh
Project Editor: Jack Litewka
Technical Editor: Marc Young

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

Copyright © 1996 by Charles Petzold

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

Petzold, Charles, 1953-
Programming Windows 95 / Charles Petzold. -- 4th ed.

p. cm.

Includes index.

ISBN 1-55615-676-6

1. Microsoft Windows (Computer file) 2. Operating systems

(Computers) I. Title.

QA76.76.O63P533 1996

005.265--dc20

95-49555

CIP

Printed and bound in the United States of America.

2 3 4 5 6 7 8 9 QM QM 1 0 9 8 7 6

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

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (206) 936-7329.

PageMaker is a trademark of Adobe Systems, Inc. Apple, LaserWriter, Lisa, Macintosh, and TrueType are registered trademarks of Apple Computer, Inc. ToolBook is a registered trademark of Asymetrix Corporation. Borland, Delphi, SideKick, and Turbo C are registered trademarks of Borland International, Inc. Corel and Design is a registered trademark of Corel Systems Corporation. DEC is a trademark of Digital Equipment Corporation. Digital Research and GEM are registered trademarks of Digital Research, Inc. ColorPro, Hewlett-Packard, and HP are registered trademarks of Hewlett-Packard Company. IBM, OS/2, and TopView are registered trademarks and Current and Graphics Assistant are trademarks of International Business Machines Corporation. Lotus and VisiCalc are registered trademarks of Lotus Development Corporation. Micrografx Designer is a trademark of Micrografx, Inc. Microsoft, Microsoft Press, Microsoft Press and Design, MS-DOS, MultiPlan, QuickC, Visual Basic, Windows, and the Windows operating system logo are registered trademarks of Microsoft Corporation. NEC is a registered trademark of NEC Corporation. PowerBuilder is a trademark of Powersoft Corporation. DESQview is a registered trademark of Quarterdeck Office Systems. Visio is a registered trademark of Visio Corporation. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. Xerox is a registered trademark of Xerox Corp. All other trademarks and service marks are the property of their respective owners.

Acquisitions Editor: Eric Stroot
Project Editor: Jack Ilewka
Technical Editor: Marc Young

Contents

Acknowledgments xi

Section I Getting Started

Chapter 1

README.TXT 3

The Programmer's Challenge ■ The Ground Rules ■ A Brief History of Windows ■ A Brief History of This Book ■ Let's Get Started

Chapter 2

Hello, Windows 95 13

The Windows Difference ■ The Graphical User Interface (GUI) ■ GUI Concepts and Rationale ■ The Consistent User Interface ■ The Multitasking Advantage ■ Memory Management ■ The Device-Independent Graphics Interface ■ The Windows Commitment ■ The Function Calls ■ Object-Oriented Programming ■ Message-Driven Architecture ■ The Window Procedure ■ Your First Windows Program ■ What's Wrong with This Program? ■ The HELLOWIN Files ■ The Make File ■ The C Source Code File ■ The Windows Function Calls ■ Uppercase Identifiers ■ New Data Types ■ Getting a Handle on Handles ■ Hungarian Notation ■ The Program Entry Point ■ Registering the Window Class ■ Creating the Window ■ Displaying the Window ■ The Message Loop ■ The Window Procedure ■ Processing the Messages ■ Playing a Sound File ■ The WM_PAINT Message ■ The WM_DESTROY Message ■ The Windows Programming Hurdles ■ Don't Call Me, I'll Call You ■ Queued and Nonqueued Messages ■ Don't Be a Pig ■ The Learning Curve

Chapter 3

Painting with Text 49

Painting and Repainting ■ The WM_PAINT Message ■ Valid and Invalid Rectangles ■ An Introduction to GDI ■ The Device Context ■ Getting a Device Context Handle: Method One ■ The Paint Information Structure ■ Getting a Device Context Handle: Method Two ■ TextOut: The Details ■ The System Font ■ The Size of a Character ■ Text Metrics: The Details ■ Formatting Text ■ Putting It All Together ■ The SYMETS1.C Window Procedure ■ Not Enough Room! ■ The Size of the Client Area ■ Scroll Bars ■ Scroll Bar Range and Position ■ Scroll Bar Messages ■ Scrolling SYMETS ■ Structuring Your Program for Painting ■ Building a Better Scroll ■ But I Don't Like to Use the Mouse

Chapter 4 Essential Graphics 87

The GDI Philosophy ■ **The Structure of GDI** The Types of Function Calls ■ The GDI Primitives ■ Other Stuff ■ **The Device Context** Getting the Handle to the Device Context ■ Getting Device Context Information ■ The DEVICAPS1 Program ■ The Size of the Device ■ Finding Out About Color ■ The Device Context Attributes ■ Saving Device Contexts ■ **Drawing Lines** The Bounding Box ■ Bezier Splines ■ Using Stock Pens ■ Creating, Selecting, and Deleting Pens ■ Filling In the Gaps ■ Drawing Modes ■ **Drawing Filled Areas** The Polygon Function and the Polygon Filling Mode ■ Brushing the Interior ■ **The Mapping Mode** Device Coordinates and Logical Coordinates ■ The Device Coordinate Systems ■ The Viewport and the Window ■ Working with MM_TEXT ■ The "Metric" Mapping Modes ■ The "Roll Your Own" Mapping Modes ■ The WHATSIZE Program ■ **Rectangles, Regions, and Clipping** Working with Rectangles ■ Random Rectangles ■ Creating and Painting Regions ■ Clipping with Rectangles and Regions ■ The CLOVER Program ■ **Paths** Creating and Rendering Paths ■ Extended Pens ■ **Bits and Bits** Color and Bitmaps ■ The Device-Independent Bitmap (DIB) ■ The DIB File ■ The Packed-DIB Memory Format ■ Displaying a DIB ■ Converting DIBs to Bitmap Objects ■ **The GDI Bitmap Object** Creating Bitmaps in a Program ■ The Monochrome Bitmap Format ■ The Color Bitmap Format ■ The Memory Device Context ■ The Mighty Bit ■ Transferring Bits with *BitBlt* ■ The Color Bitmap Format ■ Using Different ROP Codes ■ More Fun with Memory Device Contexts ■ Color Conversions ■ Mapping Mode Conversions ■ Stretching Bitmaps with *StretchBlt* ■ Brushes and Bitmaps ■ **Metafiles** Simple Use of Memory Metafiles ■ Storing Metafiles on Disk ■ **Enhanced Metafiles** Doing It Better ■ The Basic Procedure ■ Looking Inside ■ Displaying Accurate Images ■ **Text and Fonts** Simple Text Output ■ Device Context Attributes for Text ■ Using Stock Fonts ■ The Types of Fonts ■ TrueType Fonts ■ The EZFONT System ■ The Inner Workings ■ Simple Text Formatting ■ Working with Paragraphs

Section II Means of Input

Chapter 5 The Keyboard 241

Keyboard Basics Ignoring the Keyboard ■ Focus, Focus, Who's Got the Focus? ■ Keystrokes and Characters ■ **Keystroke Messages** System and Nonsystem Keystrokes ■ The *Param* Variable ■ Virtual Key Codes ■ Shift States ■ Using Keystroke Messages ■ **Enhancing SYSMETS: Adding a Keyboard Interface** Adding WM_KEYDOWN Logic ■ Sending Messages ■ **Character Messages** WM_CHAR Messages ■ Dead-Character Messages ■ **Looking at Keyboard Messages** ■ **The Caret (Not the Cursor)** The Caret Functions ■ The TYPER Program ■ **The Windows Character Sets** The OEM Character Set ■ The ANSI Character Set ■ OEM, ANSI, and Fonts ■ **Internationalization Concerns** Working with the Character Set ■ Talking with MS-DOS ■ Using the Numeric Keypad ■ The Unicode Solution for Windows NT

Chapter 6 The Mouse 287

Mouse Basics Some Quick Definitions ■ **Client-Area Mouse Messages** Simple Mouse Processing: An Example ■ Processing Shift Keys ■ Mouse Double-Clicks ■ **Nonclient-Area Mouse Messages** The Hit-Test Message ■ Messages Beget Messages ■ **Hit-Testing in Your Programs** A Hypothetical Example ■ A Sample Program ■ Emulating the Mouse with the Keyboard ■ Adding a Keyboard Interface to CHECKER ■ Using Child Windows for Hit-Testing ■ Child Windows in CHECKER ■ **Capturing the Mouse** Blocking Out a Rectangle ■ The Capture Solution ■ The BLOKOUT2 Program

Chapter 7 The Timer 327

Timer Basics The System and the Timer ■ Timer Messages Are Not Asynchronous ■ **Using the Timer: Three Methods** Method One ■ Method Two ■ Method Three ■ **Using the Timer for a Clock** Positioning and Sizing the Popup ■ Getting the Date and Time ■ Going International ■ Building an Analog Clock ■ **Windows Standard Time** ■ Animation

Chapter 8 Child Window Controls 359

The Button Class Creating the Child Windows ■ The Child Talks to Its Parent ■ The Parent Talks to Its Child ■ Push Buttons ■ Check Boxes ■ Radio Buttons ■ Group Boxes ■ Changing the Button Text ■ Visible and Enabled Buttons ■ Buttons and Input Focus ■ **Controls and Colors** System Colors ■ The Button Colors ■ The WM_CTLCOLORBTN Message ■ Owner-Draw Buttons ■ **The Scroll Bar Class** The COLORSET Program ■ The Automatic Keyboard Interface ■ Window Subclassing ■ Coloring the Background ■ Coloring the Scroll Bars and Static Text ■ **The Edit Class** The Edit Class Styles ■ Edit Control Notification ■ Using the Edit Controls ■ Messages to an Edit Control ■ **The Listbox Class** List Box Styles ■ Putting Strings in the List Box ■ Selecting and Extracting Entries ■ Receiving Messages from List Boxes ■ A Simple List Box Application ■ Listing Files ■ A Head for Windows

Section III Using Resources

Chapter 9 Icons, Cursors, Bitmaps, and Strings 421

Compiling Resources ■ **Icons and Cursors** The Image Editor ■ Getting a Handle on Icons ■ Using Icons in Your Program ■ Using Alternate Cursors ■ Bitmaps: Pictures in Pixels ■ Using Bitmaps and Brushes ■ **Character Strings** Using Character String Resources ■ Using Strings with MessageBox ■ **User-Defined Resources**

Chapter 10

Menus and Accelerators 447

Menus Menu Structure ■ The Menu Template ■ Referencing the Menu in Your Program ■ Menus and Messages ■ A Sample Program ■ Menu Etiquette ■ Defining a Menu the Hard Way ■ A Third Approach to Defining Menus ■ Floating Popup Menus ■ Using the System Menu ■ Changing the Menu ■ Other Menu Commands ■ An Unorthodox Approach to Menus ■ **Using Bitmaps in Menus** Two Methods of Creating Bitmaps for Menus ■ The Memory Device Context ■ Creating a Bitmap with Text ■ Scaling Bitmaps ■ Putting the Menu Together ■ Adding a Keyboard Interface ■ **Keyboard Accelerators** Why You Should Use Keyboard Accelerators ■ Some Rules on Assigning Accelerators ■ The Accelerator Table ■ Loading the Accelerator Table ■ Translating the Keystrokes ■ Receiving the Accelerator Messages ■ POPPAD with a Menu and Accelerators ■ Enabling Menu Items ■ Processing the Menu Options

Chapter 11

Dialog Boxes 511

Modal Dialog Boxes Creating an "About" Dialog Box ■ The Dialog Box Template ■ The Dialog Box Procedure ■ Invoking the Dialog Box ■ More on the Dialog Box Style ■ More on Defining Controls ■ A More Complex Dialog Box ■ Working with Dialog Box Controls ■ The OK and Cancel Buttons ■ Tab Stops and Groups ■ Painting on the Dialog Box ■ Using Other Functions with Dialog Boxes ■ Defining Your Own Controls ■ **Message Boxes** Popup Information ■ **Modeless Dialog Boxes** Differences Between Modal and Modeless Dialog Boxes ■ The New COLORS Program ■ HEX.ALC: Window or Dialog Box? ■ Creatively Using Control IDs ■ **The Common Dialog Boxes** POPPAD Revisited ■ Changing the Font ■ Search and Replace ■ The One Function Call Windows Program

Chapter 12

The Modern User Interface 593

Common Control Basics Library Initialization ■ Creating Common Controls ■ Common Control Styles ■ Sending Messages to Common Controls ■ Notification Messages from Common Controls ■ **Frame Window Controls** Toolbars ■ Creating a Toolbar ■ Status Bars ■ The GADGETS Program ■ **Property Sheets** Creating Property Sheets ■ Property Page Dialog Box Procedures ■ The PROPERTY Program

Chapter 13

Section IV
The Kernel and the Printer

Chapter 14

Memory Management and File I/O 719

Memory: Good, Bad, and Ugly Segmented Memory ■ Interim Solutions ■ Finally, 32 Bits and Hassle-Free Memory Allocation ■ The C Library Functions ■ Fundamental Windows 95 Memory Allocation ■ Movable Memory ■ Discardable Memory ■ Miscellaneous Functions and Flags ■ Is It Good? ■ Virtual Memory Functions ■ Heap Functions ■ **File Input/Output** The Old Way ■ The Windows 95 Difference ■ The Windows 95 File I/O Functions ■ Memory-Mapped I/O

Chapter 14

Multitasking and Multithreading 737

Modes of Multitasking Multitasking Under DOS! ■ Non-Preemptive Multitasking ■ PM and the Serialized Message Queue ■ The Multithreading Solution ■ Multithreaded Architecture ■ Thread Hassles ■ The Windows Advantage ■ New! Improved! Now with Threads! ■ **Windows 95 Multithreading** Random Rectangles Revisited ■ The Programming Contest Problem ■ The Multithreaded Solution ■ Any Problems? ■ The Benefits of Sleep ■ **Thread Synchronization** The Critical Section ■ The Mutex Object ■ **Event Signaling** The BIGJOB1 Program ■ The Event Object ■ **Thread Local Storage (TLS)**

Chapter 15

Using the Printer 783

Printing, Spooling, and the Print Functions ■ **The Printer Device Context** Getting the *CreateDC* Parameters ■ The Revised *DEVCAPS* Program ■ **The *PrinterProperties* Call** ■ Checking for *Built* Capability ■ **The Formatted Program** ■ **Printing Graphics and Text** Bare-Bones Printing ■ Canceling Printing with an Abort Procedure ■ How Windows Uses *AbortProc* ■ Implementing an Abort Procedure ■ Adding a Printing Dialog Box ■ Adding Printing to *POPPAD* ■ Handling Error Codes ■ **The Technique of Banding** Strike Up the Bands ■ Implementing Banding ■ **The Printer and Fonts**

Chapter 16

Section V
Data Exchange and Links

Chapter 16

The Clipboard 847

Simple Use of the Clipboard The Standard Clipboard Data Formats ■ Transferring Text to the Clipboard ■ Getting Text from the Clipboard ■ Opening and Closing the Clipboard ■ Using the Clipboard with Bitmaps ■ The Metafile and the Metafile Picture ■ **Beyond Simple Clipboard Use** Using Multiple Data Items ■ Delayed Rendering ■ Private Data Formats ■ **Becoming a Clipboard Viewer** The Clipboard Viewer Chain ■ Clipboard Viewer Functions and Messages ■ A Simple Clipboard Viewer

Chapter 17

Dynamic Data Exchange (DDE) 871

Basic Concepts Application, Topic, and Item ■ The Types of Conversations ■ Character Strings and Atoms ■ **A DDE Server Program** The *DDEPOP1* Program ■ The *WM_DDE_INITIATE* Message ■ The *ServerProc* Window Procedure ■ The *WM_DDE_REQUEST* Message ■ *DDEPOP1*'s *PostDataMessageFunction* ■ The *WM_DDE_ADVISE* Message ■ Updating the Items ■ The *WM_DDE_UNADVISE* Message ■ The *WM_DDE_TERMINATE* Message ■ **A DDE Client Program** Initiating the DDE Conversation ■ The *WM_DDE_DATA* Message ■ The *WM_DDE_TERMINATE* Message ■ **The DDE Management Library** Conceptual Differences ■ DDE the DDEML Way

PROGRAMMING WINDOWS 95

Chapter 18

The Multiple Document Interface (MDI) 933

The Elements of MDI ■ Windows 95 and MDI ■ The Sample Program Three Menus ■ Program Initialization
Creating the Children ■ More Frame Window Message Processing ■ The Child Document Windows ■ Cleaning
Up ■ The Power of Window Procedure

Chapter 19

Dynamic Link Libraries 959

Library Basics Library: One Word, Many Meanings ■ A Simple DLL ■ Shared Memory in DLLs The STRLIB
Library ■ The Library Entry/Exit Point ■ The STRPROG Program ■ Running STRPROG ■ Sharing Data
Among STRPROG Instances ■ Some Library Restrictions ■ Dynamic Linking Without Imports ■ Resource-Only
Libraries

Chapter 20

What's This Thing Called OLE? 987

OLE Basics Connecting to the OLE Libraries ■ Deciphering Result Codes ■ Component Object Model (COM)
Interfaces ■ *Unknown* Services ■ Is OLE a Client/Server Specification? ■ A Private Component Server
IMAILLOC.DLL ■ Those Funky Macros ■ Services Provided by *Unknown* ■ A Private Component Client ■
A Public Component Server Role of the Registry ■ Mechanics of CLSID Generation and Use ■ The Class
Factory Component ■ Controlling Server Lifetime ■ A Public Component Client ■ FINAL WORDS

Index 1061

Dynamic Link Libraries

Dynamic link libraries (also called DLLs, dynamic libraries, "dynalink" libraries, or library modules) are one of the most important structural elements of Windows. Most of the disk files associated with Windows are either program modules or dynamic link library modules. So far we've been writing Windows programs; now it's time to take a stab at writing dynamic link libraries. Many of the principles you've learned in writing programs are also applicable to writing these libraries, but there are some important differences.

LIBRARY BASICS

As you've seen, a Windows program is an executable file that generally creates one or more windows and uses a message loop to receive user input. Dynamic link libraries are generally not directly executable, and they usually do not receive messages. They are separate files containing functions that can be called by programs and other DLLs to perform certain jobs. A dynamic link library is brought into action only when another module calls one of the functions in the library.

The term *dynamic linking* refers to the process that Windows uses to link a function call in one module to the actual function in the library module. *Static linking* occurs during program development when you link various object (.OBJ) modules, run-time library (.LIB) files, and usually a compiled resource (.RES) file to create a Windows .EXE file. Dynamic linking instead occurs at run time.

KERNEL32.DLL, USER32.DLL, and GDI32.DLL, the various driver files such as KEYBOARD.DRV, SYSTEM.DRV, and MOUSE.DRV, and the video and printer drivers are all dynamic link libraries. These are libraries that all Windows programs can use.

Some dynamic link libraries (such as font files) are termed "resource-only." They contain only data (usually in the form of resources) and no code. Thus, one purpose of dynamic link libraries is to provide functions and resources that can be used by many different programs. In a conventional operating system, only the operating system itself contains routines that other programs can call on to do a job. In Windows, the process of one module calling a function in another module is generalized. In effect, by writing a dynamic link library, you are writing an extension to Windows. Or you can think of dynamic link libraries (including those that make up Windows) as extensions to your program.

Although a dynamic link library module may have any extension (such as .EXE or .FON), the standard extension in Microsoft Windows 95 is .DLL. Only dynamic link libraries with the extension .DLL will be loaded automatically by Windows. If the file has another extension, the program must explicitly load the module using the *LoadLibraryEx* function.

You'll generally find that dynamic link libraries make most sense in the context of a large application. For instance, suppose you write a large accounting package for Windows that consists of several different programs. You'll probably find that these programs use many common routines. You could put these common routines in a normal object library (with the extension .LIB) and add them to each of the program modules during static linking with LINK. But this approach is wasteful, because each of the programs in this package contains identical code for the common routines. Moreover, if you change one of the routines in this library, you'll have to relink all the programs that use the changed routine. If, however, you put these common routines in a dynamic link library called (for instance) ACCOUNT.DLL, you've solved both problems. Only the library module need contain the routines required by all the programs (thus requiring less disk space for the files and less memory space when running two or more of the applications simultaneously), and you can make changes to the library module without relinking any of the individual programs.

Dynamic link libraries can themselves be viable products. For instance, suppose you write a collection of three-dimensional drawing routines and put them in a dynamic link library called GDI3.DLL. If you then interest other software developers in using your library, you can license it to be included with their graphics programs. A user who has several of these programs would need only one GDI3.DLL file.

Library: One Word, Many Meanings

Part of the confusion surrounding dynamic link libraries results from the appearance of the word library in several different contexts. Besides dynamic link libraries, we'll also be talking about "object libraries" and "import libraries."

An object library is a file with the extension .LIB containing code that is added to your program's .EXE file when you run the linker during static linking. For example, in Microsoft Visual C++, the normal C runtime object library that you link with your program is called IIBC.LIB.

An import library is a special form of an object library file. Like object libraries, import libraries have the extension .LIB and are used by the linker to resolve function calls in your source code. However, import libraries contain no code. Instead, they provide the linker with information necessary to set up relocation tables within the .EXE file for dynamic linking. The KERNEL32.LIB, USER32.LIB, and GDI32.LIB files included with the Microsoft compiler are import libraries for Windows functions. If you call *Rectangle* in a program, GDI32.LIB tells LINK that this function is in the GDI32.DLL dynamic link library. This information goes into the .EXE file so that Windows can perform dynamic linking with the GDI32.DLL dynamic link library when your program is executed.

Object libraries and import libraries are used only during program development. Dynamic link libraries are used during run time. A dynamic link library must be present on the disk when a program is run that uses the library. When Windows 95 needs to load a dynamic link library module before running a program that requires it, the library file must be stored in the directory containing the .EXE program, the current directory, the Windows system directory, the Windows directory, or a directory accessible through the PATH string in the MS-DOS environment. (The directories are searched in that order.)

A SIMPLE DLL

As usual, we'll begin very simply. Figure 19-1 shows the necessary source code for a dynamic link library called EDRLIB that contains only one function. The "EDR" of this filename stands for "easy drawing routine," and you can easily add other functions to it that simplify the drawing functions in your applications. The single function in EDRLIB is called *EdrCenterText*, and it simply centers a NULL-terminated text string in a rectangle, much like *DrautText* but without as many parameters.

```
EDRLIB.MAK
#-----
# EDRLIB.MAK make file
#-----
edrlib.dtl : edrlib.obj
$(LINKER) $(OFLAGS) -OUT:edrlib.dtl edrlib.obj $(GUILIBS)
edrlib.obj : edrlib.c edrlib.h
$(CC) $(CFLAGS) edrlib.c
```

Figure 19-1. The EDRLIB library.

(continued)

EDRLIB.H

```

/*-----
EDRLIB.H header file
-----*/

#define EXPORT extern "C" __declspec (dllexport)

EXPORT BOOL CALLBACK EdrCenterText (HDC, PRECT, PSTR) ;

```

EDRLIB.C

```

/*-----
EDRLIB.C -- Easy Drawing Routine Library module
(c) Charles Petzold, 1996
-----*/

#include <windows.h>
#include <string.h>
#include "edrlib.h"

int WINAPI DllMain (HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved)
{
    return TRUE ;
}

EXPORT BOOL CALLBACK EdrCenterText (HDC hdc, PRECT prc, PSTR pString)
{
    int iLength ;
    SIZE size ;

    iLength = strlen (pString) ;

    GetTextExtentPoint32 (hdc, pString, iLength, &size) ;

    return TextOut (hdc, (prc->right - prc->left - size.cx) / 2,
                    (prc->bottom - prc->top - size.cy) / 2,
                    pString, iLength) ;
}

```

As you can see, the make file has two differences from the make files we've been using for our applications. First, in the link command line, I've substituted the environment variable `DLLFLAGS` for `GUIFLAGS`. This was included in the `MSC.BAT` file I introduced in Chapter 1 and simply includes the link command-line parameter `-DLL`. This parameter tells

the linker that the resultant file will be a dynamic link library. Also, you'll notice that instead of creating a `.EXE` file, the make file creates a file named `EDRLIB.DLL`.

The `EDRLIB.H` file first defines a macro constant named `EXPORT`:

```
#define EXPORT extern "C" __declspec (dllexport)
```

Using the `EXPORT` keyword when defining functions in your dynamic link library will tell the linker that the functions are available for use by other programs. The `EdrCenterText` function is defined in the header file using this `EXPORT` definition. The function is also defined using the `CALIBACK` constant, just like a window procedure.

`EDRLIB.C` also looks a little different from normal Windows C files. It includes a function named `DllMain` rather than `WinMain`. This function is used to perform initialization and de-initialization, as I'll discuss later in this chapter. For our purposes, all we need do right now is return `TRUE`. And finally, `EDRLIB.C` contains the `EdrCenterText` function.

When you run

```
MMAKE EDRLIB.MAK
```

on a command line, you'll create `EDRLIB.DLL`, the dynamic link library. You'll also find two other new files. `EDRLIB.LIB` is an import library that we'll use shortly. `EDRLIB.EXP` is an export library that is created as a residue of the linking process. You may delete it.

Does `EDRLIB.DLL` work? Well, let's try it out. `EDRTEST`, shown in Figure 19-2, uses `EDRLIB.DLL` to center a text string in its client area.

EDRTEST.MAK

```

#-----
# EDRTEST.MAK make file
#-----

edrttest.exe : edrttest.obj edrlib.lib
$(LINKER) $(GUIFLAGS) -OUT:edrttest.exe edrttest.obj edrlib.lib $(GUILIBS)

edrttest.obj : edrttest.c edrlib.h
$(CC) $(CFLAGS) edrttest.c

```

EDRTEST.C

```

/*-----
EDRTEST.C -- Program using EDRLIB dynamic link library
(c) Charles Petzold, 1996
-----*/

```

Figure 19-2. The `EDRTEST` program.

(continued)

```

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

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
    PSTR szCmdLine, int iCmdShow)
{
    static char szAppName[] = "StrProg" ;
    HWND hwnd ;
    MSG msg ;
    WNDCLASSEX wndclass ;

    wndclass.cbSize = sizeof (wndclass) ;
    wndclass.style = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = szAppName ;
    wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION) ;

    RegisterClassEx (&wndclass) ;

    hwnd = CreateWindow (szAppName, "DLL Demonstration Program",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT,
        CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

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

    return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)

```

(continued)

```

{
    HDC hdc ;
    PAINTSTRUCT ps ;
    RECT rect ;

    switch (iMsg)
    {
        case WM_PAINT :
            hdc = BeginPaint (hwnd, &ps) ;
            GetClientRect (hwnd, &rect) ;
            EdrCenterText (hdc, &rect,
                "This string was displayed by a DLL") ;
            EndPaint (hwnd, &ps) ;
            return 0 ;

        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
    }

    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

```

This looks like a normal Windows program and it certainly is. Note, however, that `EDRTEST.C` includes `EDRLIB.H` to define the *EdrCenterText* function that the program calls during the `WM_PAINT` message. Also notice that `EDRTEST.MAK` includes the `EDRLIB.LIB` file in the link statement. That import library provides the information `LINK` needs to set up a reference in the `EDRTEST.EXE` file to the `EDRLIB.DLL` file.

It is vital to understand that the *EdrCenterText* code is not included in the `EDRTEST.EXE` file. Instead, there is simply a reference to the `EDRLIB.DLL` file and the *EdrCenterText* function. `EDRTEST.EXE` requires `EDRLIB.DLL` to run.

Including `EDRLIB.H` in the `EDRTEST.C` source code file is equivalent to including `WINDOWS.H`. Including `EDRLIB.LIB` in the link command line is equivalent to including all the Windows import libraries (such as `USER32.LIB`), which are listed in the `GULIBS` environment variable referenced by the make file. When your program runs, it links with `EDLIB.DLL` in the same way it links with `USER32.DLL`. Congratulations! You've just created an extension to Windows 95!

A few words on the subject of dynamic link libraries before we continue... Although I've just categorized a DLL as an extension to Windows 95, it is also an extension to your application program. Everything the DLL does is done on behalf of the application. For example, all memory it allocates is owned by the application. Any windows it creates are owned by the application. Any files it opens are owned by the application. Multiple

applications can use the same DLL simultaneously, but under Windows 95 these applications are shielded from interfering with each other. However, if you write a DLL with functions that can be called from multiple threads of a program, you should compile the DLL with the `FLAGSMT` environment variable rather than `FLAGS`.

Because code is nonwritable, different processes can share the same code in a dynamic link library. The data maintained by a DLL, however, is different for each process. Each process has its own address space for any data the DLL uses. Sharing memory among processes requires extra work (as we'll see).

SHARED MEMORY IN DLLS

It's very nice that Windows 95 isolates applications that are using the same dynamic link libraries at the same time. However, sometimes it's not preferable. You may want to write a DLL that contains some memory that can be shared among various applications, or perhaps multiple instances of the same application. This involves using shared memory (actually a memory-mapped file), which I mentioned in Chapter 13 but promised I'd demonstrate later.

Let's examine how this works with a program called `STRPROC` ("string program") and the dynamic link library called `STRLIB` ("string library"). `STRLIB` has three exported functions that `STRPROC` calls. Just to make this interesting, one of the functions in `STRLIB` uses a call-back function defined in `STRPROC`.

`STRLIB` is a dynamic link library module that stores and sorts up to 256 character strings. The strings are capitalized and maintained by shared memory in `STRLIB`. `STRPROC` can use `STRLIB`'s three functions to add strings, delete strings, and obtain all the current strings from `STRLIB`. The program has two menu items (Enter and Delete) that invoke dialog boxes to add and delete these strings. `STRPROC` lists all the current strings stored by `STRLIB` in `STRPROC`'s client area.

This function defined in `STRLIB` adds a string to `STRLIB`'s shared memory:

```
EXPORT BOOL CALLBACK AddString (PSTR pStringIn)
```

The parameter *pStringIn* is a pointer to the string. The string is capitalized within the *AddString* function. If an identical string already exists in `STRLIB`'s list of strings, this function adds another copy of the string. *AddString* returns `TRUE` (nonzero) if it is successful and `FALSE` (0) otherwise. A `FALSE` return value can result if the string has a length of 0, if memory could not be allocated to store the string, or if 256 strings are already stored.

This `STRLIB` function deletes a string from `STRLIB`'s shared memory:

```
EXPORT BOOL CALLBACK DeleteString (PSTR pStringIn)
```

Again, the parameter *pStringIn* is a pointer to the string. If more than one string matches, only the first is removed. *DeleteString* returns `TRUE` (nonzero) if it is successful and `FALSE` (0) otherwise. A `FALSE` return value indicates that the length of the string is 0 or that a matching string could not be found.

This `STRLIB` function uses a call-back function located in the calling program to enumerate the strings currently stored in `STRLIB`'s shared memory:

```
EXPORT int CALLBACK GetStrings (PSTR pFindStrCallback, Pvoid pParam)
```

The call-back function must be defined as follows:

```
EXPORT BOOL CALLBACK GetStrCallback (PSTR pString, Pvoid pParam)
```

The *pFindStrCallback* parameter to *GetStrings* points to the call-back function. *GetStrings* calls *GetStrCallback* once for each string or until the call-back function returns `FALSE` (0). *GetStrings* returns the number of strings passed to the call-back function. The *pParam* parameter is a pointer to programmer-defined data.

The STRLIB Library

Figure 19-3 shows the three files necessary to create the `STRLIB.DLL` dynamic link library module.

STRLIB.MAK

```
#-----
# STRLIB.MAK make file
#-----

strlib.dll : strlib.obj
$(LINKER) $(DLLFLAGS) -SECTION:shared,rws -OUT:strlib.dll \
strlib.obj $(BUILDS)

strlib.obj : strlib.c strlib.h
$(CC) $(CFLAGS) strlib.c
```

STRLIB.H

```
/*-----
STRLIB.H header file
-----*/

typedef BOOL (CALLBACK *PSTRCB) (PSTR, Pvoid);
#define MAX_STRINGS 256

#define EXPORT extern "C" __declspec (dllexport)

EXPORT BOOL CALLBACK AddString (PSTR);
EXPORT BOOL CALLBACK DeleteString (PSTR);
EXPORT int CALLBACK GetStrings (PSTRCB, Pvoid);
```

Figure 19-3. The `STRLIB` library.

(continued)

STRLIB.C

```

/*-----
   STRLIB.C -- Library module for STRPROG program
   (c) Charles Patzold, 1996
   -----*/

#include <Windows.h>
#include "strlib.h"

#pragma data_seg ("shared")

PSTR pszStrings[MAX_STRINGS] = { NULL };
int iTotal = 0;

#pragma data_seg ()

int WINAPI DllMain (HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved)
{
    int i;

    switch (fdwReason)
    {
        // Nothing to do when process (or thread) begins

        case DLL_PROCESS_ATTACH :
        case DLL_THREAD_ATTACH :
        case DLL_THREAD_DETACH :
            break;

        // When process terminates, free any remaining blocks

        case DLL_PROCESS_DETACH :
            for (i = 0; i < iTotal; i++)
                UnmapViewOfFile (pszStrings[i]);
            break;
    }

    return TRUE;
}

EXPORT BOOL CALLBACK AddString (PSTR pStringIn)
{
    HANDLE hString;
    PSTR pString;

```

(continued)

```

int i, iLength, iCompare;

if (iTotal == MAX_STRINGS - 1)
    return FALSE;

iLength = strlen (pStringIn);
if (iLength == 0)
    return FALSE;

hString = CreateFileMapping ((HANDLE) -1, NULL, PAGE_READWRITE,
                             0, 1 + iLength, NULL);

if (hString == NULL)
    return FALSE;

pString = (PSTR) MapViewOfFile (hString, FILE_MAP_WRITE, 0, 0, 0);
strcpy (pString, pStringIn);
AnsiUpper (pString);

for (i = iTotal; i > 0; i--)
{
    iCompare = strcmpf (pStringIn, pszStrings[i - 1]);

    if (iCompare >= 0)
        break;

    pszStrings[i] = pszStrings[i - 1];
}

pszStrings[i] = pString;

iTotal++;
return TRUE;
}

EXPORT BOOL CALLBACK DeleteString (PSTR pStringIn)
{
    int i, j, iCompare;

    if (0 == strlen (pStringIn))
        return FALSE;

    for (i = 0; i < iTotal; i++)
    {
        iCompare = lstrcmpf (pszStrings[i], pStringIn);

```

(continued)

```

    if (iCompare == 0)
        break ;
    }

    // If given string not in list, return without taking action
    if (i != iTotal)
        return FALSE ;

    // Else free memory occupied by the string and adjust list downward
    UnmapViewOfFile (pszStrings[i]) ;

    for (j = i ; j < iTotal ; j++)
        pszStrings[j] = pszStrings[j + 1] ;

    pszStrings[iTotal--] = NULL ;    // Destroy unused pointer
    return TRUE ;
}

EXPORT int CALLBACK GetStrings (PSTRCB pfnGetStringCallback, PVOID pParam)
{
    BOOL bReturn ;
    int i ;

    for (i = 0 ; i < iTotal ; i++)
    {
        bReturn = pfnGetStringCallback (pszStrings[i], pParam) ;

        if (bReturn == FALSE)
            return i + 1 ;
    }

    return iTotal ;
}

```

The Library Entry/Exit Point

As you can see in `STRLIB.C`, we've now made some use of *DllMain*. This function is called when the library first begins and when it terminates. Although I also included a *DllMain* function in `EDRLIB.C`, it's not really necessary; a function that performs identically would have been included by the linker by default.

The first parameter to *DllMain* is the instance handle of the library. If your library uses resources that require an instance handle (such as *DialogBox*), you should save *hInstance* as a global variable. The last parameter to *DllMain* is reserved by the system.

The *fdwReason* parameter can be one of four values that indicate why Windows 95 is calling the *DllMain* function. In the following discussion, keep in mind that a single program can be loaded multiple times and run together under Windows. Each time a program is loaded, it is considered a separate process.

An *fdwReason* value of `DLL_PROCESS_ATTACH` indicates that the dynamic link library has been mapped into the address space of a process. This is a cue for the library to do any initialization tasks it requires to service subsequent requests from the process. Such initialization may include memory allocation, for example. During the time that a process is running, *DllMain* is called with a `DLL_PROCESS_ATTACH` parameter only once during the lifetime of that process. Any other process using the same dynamic link library causes another call to *DllMain* with a `DLL_PROCESS_ATTACH` parameter, but that's on behalf of the new process.

If the initialization is successful, *DllMain* should return a nonzero value. Returning zero will cause Windows to not run the program.

When *fdwReason* has a value of `DLL_PROCESS_DETACH`, it means that the dynamic link library is no longer needed by the process. This provides an opportunity for the library to clean up after itself. Under Windows 95 this is not strictly necessary, but it's a good programming practice.

Similarly, when *DllMain* is called with an *fdwReason* parameter of `DLL_THREAD_ATTACH`, it means that an attached process has created a new thread. When the thread terminates, Windows calls *DllMain* with an *fdwReason* parameter of `DLL_THREAD_DETACH`. Be aware that it's possible to get a `DLL_THREAD_DETACH` call without an earlier `DLL_THREAD_ATTACH` call if the dynamic link library is attached to a process after the thread has been created.

The thread still exists when *DllMain* is called with a parameter of `DLL_THREAD_DETACH`. The dynamic link library can even send the thread messages during this process. But it shouldn't use *PostMessage*, because the thread will probably be gone before the message is retrieved.

Aside from the *DllMain* function, `STRLIB` contains only the three functions that it will export to be used by other programs. All these functions are defined as `EXPORT`. This causes `LINK` to list them in the `STRLIB.LIB` import library.

The STRPROG Program

The `STRPROG` program, shown in Figure 19-4, is fairly straightforward. The two menu options (Enter and Delete) invoke dialog boxes that allow you to enter a string. `STRPROG` then calls *AddString* or *DeleteString*. When the program needs to update its client area, it calls *GetStrings* and uses the function *GetStringCallback* to list the enumerated strings.

STRPROG.MAK

```

#-----
# STRPROG.MAK make file
#-----
strprog.exe : strprog.obj strprog.res strlib.lib
$(LINKER) $(GUILFLAGS) -OUT:strprog.exe strprog.obj strprog.res \
strlib.lib $(GUILIBS)

strprog.obj : strprog.c strprog.h strlib.h
$(CC) $(CFLAGS) strprog.c

strprog.res : strprog.rc strprog.h
$(RC) $(RCFLAGS) strprog.rc

```

STRPROG.C

```

/*-----
STRPROG.C -- Program using STRLIB dynamic link library
(c) Charles Petzold, 1996
-----*/

#include <windows.h>
#include <string.h>
#include "strprog.h"
#include "strlib.h"

#define MAXLEN 32
#define WM_DATACHANGE WM_USER

typedef struct
{
    HDC hdc ;
    int xText ;
    int yText ;
    int xStart ;
    int yStart ;
    int xIncr ;
    int yIncr ;
    int xMax ;
    int yMax ;
} CBPARAM ;

```

Figure 19-4. The STRPROG program.

(continued)

```

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char szAppName[] = "StrProg" ;
char szString[MAXLEN] ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    HWND hwnd ;
    MSG msg ;
    WNDCLASSEX wndClass ;

    wndClass.cbSize = sizeof (wndClass) ;
    wndClass.style = CS_HREDRAW | CS_VREDRAW ;
    wndClass.lpfnWndProc = WndProc ;
    wndClass.cbClsExtra = 0 ;
    wndClass.cbWndExtra = 0 ;
    wndClass.hInstance = hInstance ;
    wndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
    wndClass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
    wndClass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndClass.lpszMenuName = szAppName ;
    wndClass.lpszClassName = szAppName ;
    wndClass.hIconSm = LoadIcon (NULL, IDI_APPLICATION) ;

    RegisterClassEx (&wndClass) ;

    hwnd = CreateWindow (szAppName, "DLL Demonstration Program",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}

BOOL CALLBACK DlgProc (HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)

```

(continued)

```

{
    switch (iMsg)
    {
        case WM_INITDIALOG :
            SendDlgItemMessage (hDlg, IDD_STRING, EM_LIMITTEXT,
                               MAXLEN - 1, 0);
            return TRUE;

        case WM_COMMAND :
            switch (wParam)
            {
                case IDOK :
                    GetDlgItemText (hDlg, IDD_STRING, szString, MAXLEN);
                    EndDialog (hDlg, TRUE);
                    return TRUE;

                case IDCANCEL :
                    EndDialog (hDlg, FALSE);
                    return TRUE;
            }
            return FALSE;
    }
}

800L CALLBACK EnumCallBack (HWND hwnd, LPARAM lParam)
{
    char szClassName[16];
    GetClassName (hwnd, szClassName, sizeof (szClassName));
    if (0 == strcmp (szClassName, szAppName))
        SendMessage (hwnd, WM_DATACHANGE, 0, 0);
    return TRUE;
}

800L CALLBACK GetStrCallBack (PSTR pString, CBPARAM *pcbp)
{
    TextOut (pcbp->hdc, pcbp->Text, pcbp->yText,
             pString, strlen (pString));
    if ((pcbp->yText += pcbp->yIncr) > pcbp->yMax)
    {
        pcbp->yText = pcbp->yStart;
        if ((pcbp->xText += pcbp->xIncr) > pcbp->xMax)
    }
}

```

(continued)

```

    }
    return FALSE;
}

return TRUE;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    static HINSTANCE hInst;
    static int cxChar, cyChar, cxClient, cyClient;
    CBPARAM cbParam;
    HDC hdc;
    PAINTSTRUCT ps;
    TEXTMETRIC tm;

    switch (iMsg)
    {
        case WM_CREATE :
            hInst = ((LPCREATESTRUCT) lParam)->hInstance;
            hdc = GetDC (hwnd);
            GetTextMetrics (hdc, &tm);
            cxChar = (int) tm.tmAveCharWidth;
            cyChar = (int) (tm.tmHeight + tm.tmExternalLeading);
            ReleaseDC (hwnd, hdc);

            return 0;

        case WM_COMMAND :
            switch (wParam)
            {
                case IDM_ENTER :
                    if (DialogBox (hInst, "EnterDlg", hwnd, &DlgProc))
                    {
                        if (AddString (szString))
                            EnumWindows (&EnumCallBack, 0);
                        else
                            MessageBeep (0);
                    }
                    break;

                case IDM_DELETE :
                    if (DialogBox (hInst, "DeleteDlg", hwnd, &DlgProc))
                    {
                        if (DeleteString (szString))
                            EnumWindows (&EnumCallBack, 0);
                    }
            }
    }
}

```

(continued)

```

        else
            MessageBeep (0);
        }
        break;
    }
    return 0;
}

case WM_SIZE :
    cxClient = (int) LOWORD (lParam);
    cyClient = (int) HIWORD (lParam);
    return 0;

case WM_DATACHANGE :
    InvalidateRect (hwnd, NULL, TRUE);
    return 0;

case WM_PAINT :
    hdc = BeginPaint (hwnd, &ps);

    cbparam.hdc = hdc;
    cbparam.xText = cbparam.xStart = cxChar;
    cbparam.yText = cbparam.yStart = cyChar;
    cbparam.xIncr = cxChar * MAXLEN;
    cbparam.yIncr = cyChar;
    cbparam.xMax = cbparam.xIncr * (1 + cxClient / cbparam.xIncr);
    cbparam.yMax = cyChar * (cyClient / cyChar - 1);

    GetStrings ((HSTRCB) GetStrCallback, (PVOID) &cbparam);

    EndPaint (hwnd, &ps);
    return 0;

case WM_DESTROY :
    PostQuitMessage (0);
    return 0;
}

return DefWindowProc (hwnd, lMsg, wParam, lParam);
}

```

STRPROG.RC

```

/*-----
STRPROG.RC resource script
-----*/

```

(continued)

```

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

StrProg MENU
{
    MENUITEM "&Enter!", IDM_ENTER
    MENUITEM "&Delete!", IDM_DELETE
}

EnterDlg DIALOG 24, 24, 190, 44
    STYLE WS_POPUP | WS_DLGFRAME
{
    LTEXT "&Enter:", 0, 4, 8, 24, 8
    EDITTEXT IDD_STRING, 32, 6, 154, 12
    DEFUSHBUTTON "OK", 44, 24, 32, 14
    PUSHBUTTON "Cancel", 114, 24, 32, 14
}

DeleteDlg DIALOG 24, 24, 190, 44
    STYLE WS_POPUP | WS_DLGFRAME
{
    LTEXT "&Delete:", 0, 4, 8, 28, 8
    EDITTEXT IDD_STRING, 36, 6, 150, 12
    DEFUSHBUTTON "OK", 44, 24, 32, 14
    PUSHBUTTON "Cancel", 114, 24, 32, 14
}

```

STRPROG.H

```

/*-----
STRPROG.H header file
-----*/

#define IDM_ENTER 1
#define IDM_DELETE 2
#define IDD_STRING 0x10

```

STRPROG.C includes the STRPROG.H header file, which simply defines the constants used in the STRPROG.RC resource script. It also includes the STRLIB.H header file; this defines the three functions in STRLIB that STRPROG will use.

Running STRPROG

Once you've created STRLIB.DLL and STRPROG.EXE, you're ready to run STRPROG. Before you do so, be sure that STRLIB.DLL is in the current directory or a directory that is accessible

by Windows (as discussed earlier). Windows must be able to load STRLIB.DLL when you execute STRPROG. If Windows can't find STRLIB.DLL, it will display a message box telling you about the problem.

When you execute STRPROG.EXE, Windows performs fix-ups to functions in external library modules. Many of these functions are in the normal Windows dynamic link libraries. But Windows also sees that the program calls three functions from STRLIB, so Windows loads the STRLIB.DLL file into memory and calls STRLIB's initialization routine in STRLIB. You can then use STRPROG to add and delete strings from STRLIB's internal table. STRPROG's client area shows the strings currently in the table.

The calls from STRPROG to the *AddString*, *DeleteString*, and *GetString* functions in STRLIB are very efficient and have no more overhead than calls to any other external module. In fact, the link between STRPROG and STRLIB is as efficient as if the three functions in STRLIB were simply in STRPROG itself. So what? you say. Why do I have to make this a dynamic link library? Can't I include these three routines in STRPROG.EXE?

Well, you could. In one sense, STRLIB is nothing more than an extension of STRPROG. However, you may be interested to see what happens when you execute a second instance of STRPROG. STRLIB stores the character strings and their pointers in shared memory, which lets all instances of STRPROG share this data. Let's look at how it's done.

Sharing Data Among STRPROG Instances

Windows erects a wall around the address space of a Win32 process. Normally, data in an address space is private, invisible to other processes. But running multiple instances of STRPROG shows that STRLIB has no trouble sharing its data with all instances of the program. When you add or delete a string in a STRPROG window, the change is immediately reflected in the other windows.

STRLIB shares two types of data: the strings and the pointers to the strings. For the sake of illustration, it uses a different sharing method for each type. We've already encountered one method in Chapter 13. STRLIB stores each string in a memory-mapped file, making the string visible to all processes.

As for the string pointers, STRLIB keeps them in a special section of memory that it designates as shared:

```
#pragma data_seg ("shared")
PSTR pszStrings[MAX_STRINGS] = { NULL };
int iTotal = 0;
#pragma data_seg (
```

The first *#pragma* statement creates the data section, here named *shared*. You can name the section whatever you wish, though the linker sees only the first eight characters. All initialized variables after the *#pragma* go into the *shared* section. The second *#pragma*

statement marks the end of the section. It's important to specifically initialize the variables; otherwise the compiler puts them in the normal uninitialized section rather than in *shared*. The linker has to be told about *shared*. On the linker command line, specify the *-SECTION:shared,rws* parameter like this:

```
-SECTION:shared,rws
```

The "rws" letters indicate the section has read, write, and shared attributes. Now all instances of STRPROG see one instance of the strings and pointers. The *EnumCallback* function in STRPROG serves to notify all STRPROG's instances that the contents of STRLIB's data segments have changed. *EnumWindows* causes Windows to call *EnumCallback* with handles to all parent windows. *EnumCallback* then checks to see if the class name of each window equals "StrProg"; if it does, the function sends the window a privately defined WM_DATACHANGE message. And you can easily imagine an enhanced version of STRLIB managing a database that is shared by several instances of the same program or by single instances of different programs.

SOME LIBRARY RESTRICTIONS

I mentioned earlier that a dynamic link library module doesn't receive messages. However, a library module can call *GetMessage* and *PeekMessage*. The messages the library pulls from the queue with these functions are actually messages for the program that called the library function. In general, the library works on behalf of the program calling it, a rule that holds for most Windows functions that a library calls.

A dynamic link library can load resources (such as icons, strings, and bitmaps) either from the library file or from the file of the program that calls the library. The functions that load resources require an instance handle. If the library uses its own instance handle (which is passed to the library during initialization), then the library can obtain resources from its own file. To load resources from the calling program's .EXE file, the library function requires the instance handle of the program calling the function.

Registering window classes and creating windows in a library can be a little tricky. Both the window class structure and the *CreateWindow* call require an instance handle. Although you can use the library module's instance handle in creating the window class and the window, the window messages still go through the message queue of the program calling the library when the library creates the window. If you must create window classes and windows within a library, it's probably best to use the calling program's instance handle.

Because messages for modal dialog boxes are retrieved outside a program's message loop, you can create a modal dialog box in a library by calling *DialogBox*. The instance handle can be that of the library, and the *hWndParent* parameter to *DialogBox* can be set to NULL.

DYNAMIC LINKING WITHOUT IMPORTS

Rather than have Windows perform dynamic linking when your program is first loaded into memory, you can link a program with a library module while the program is running. For instance, you would normally call the *Rectangle* function like this:

```
Rectangle(hdc, xLeft, yTop, xRight, yBottom);
```

This works because the program has been linked with the GDI32.LIB import library, which supplied the address of *Rectangle*.

You can also call *Rectangle* in a very roundabout manner. You first use *typedef* to define a function type for *Rectangle*:

```
typedef BOOL (WINAPI *PPIRECT) (HDC, int, int, int, int);
```

You then define two variables:

```
HANDLE hLibrary;  
PPIRECT pfnRectangle;
```

Now you set *hLibrary* to the handle of the library and *pfnRectangle* to the address of the *Rectangle* function:

```
hLibrary = LoadLibrary("GDI32.DLL");  
pfnRectangle = (PPIRECT) GetProcAddress(hLibrary, "Rectangle");
```

The *LoadLibrary* function returns NULL if the library file can't be found or some other error occurs. Now you can call the function and then free the library:

```
pfnRectangle(hdc, xLeft, yTop, xRight, yBottom);  
FreeLibrary(hLibrary);
```

Although this technique of run-time dynamic linking doesn't make much sense for the *Rectangle* function, it will definitely come in handy when you don't know the name of the library module until run time.

The code above uses the *LoadLibrary* and *FreeLibrary* functions. Windows maintains "reference counts" for all library modules. *LoadLibrary* causes the reference count to be incremented. The reference count is also incremented when Windows loads any program that uses the library. *FreeLibrary* causes the reference count to be decremented, as does the termination of an instance of a program that uses this library. When the reference count is 0, Windows can discard the library from memory, because the library is no longer needed.

RESOURCE-ONLY LIBRARIES

Any function in a dynamic link library that Windows programs or other libraries can use must be exported. However, a dynamic link library need not contain any exported functions. What would such a DLL contain? The answer is resources.

Let's say you're working on a Windows application that requires a number of bitmaps. Normally you would list these in the resource script of the program and load them into memory with the *LoadBitmap* function. But perhaps you want to create several sets of bitmaps, each set customized for one of the major display adapters used with Windows. It would make most sense to store these different sets of bitmaps in different files, because a user would need only one set of bitmaps on the fixed disk. These files are resource-only libraries.

Figure 19-5 shows how to create a resource-only library file called BITLIB.DLL that contains nine bitmaps. The BITLIB.RC file lists all the separate bitmap files and assigns each one a number. To create BITLIB.DLL, you need nine bitmaps named BITMAP1.BMP, BITMAP2.BMP, and so forth. You can use the bitmaps provided on the companion CD-ROM or create them yourself in the Windows PAINT program.

BITLIB.MAK

```
#-----  
# BITLIB.MAK make file  
#-----  
  
bitlib.dll : bitlib.obj bitlib.res  
$(LINKER) $(DLFLAGS) -OUT:bitlib.dll bitlib.obj bitlib.res $(GUILIBS)  
  
bitlib.obj : bitlib.c  
$(CC) $(CFLAGS) bitlib.c  
  
bitlib.res : bitlib.rc  
$(RC) $(RCFLAGS) bitlib.rc
```

BITLIB.C

```
/*  
 BITLIB.C -- Code entry point for BITLIB dynamic link library  
 (c) Charles Petzold, 1996  
-----*/  
  
#include <windows.h>
```

Figure 19-5. The BITLIB library.

(continued)

```
int WINAPI DllMain (HINSTANCE hinstance, DWORD fdwReason, PVOID pvReserved)
{
    return TRUE ;
}
```

```
BITLIB.RC
/*-----
BITLIB.RC resource script
-----*/

1 BITMAP bitmap1.bmp
2 BITMAP bitmap2.bmp
3 BITMAP bitmap3.bmp
4 BITMAP bitmap4.bmp
5 BITMAP bitmap5.bmp
6 BITMAP bitmap6.bmp
7 BITMAP bitmap7.bmp
8 BITMAP bitmap8.bmp
9 BITMAP bitmap9.bmp
```

The SHOWBIT program, shown in Figure 19-6, reads the bitmap resources from BITLIB and draws them in the upper-left corner of the client area. You can cycle through the bitmaps by pressing a key on the keyboard.

```
SHOWBIT.MAK
#-----
# SHOWBIT.MAK make file
#-----

showbit.exe : showbit.obj
$(LINKER) $(GUILIBS) -OUT:showbit.exe showbit.obj $(GUILIBS)

showbit.obj : showbit.c
$(CC) $(CFLAGS) showbit.c
```

```
SHOWBIT.C
/*-----
SHOWBIT.C --- Shows bitmaps in BITLIB dynamic link library
(c) Charles Petzold, 1996
-----*/
```

Figure 19-6. The SHOWBIT program.

(continued)

```
#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
    PSTR szCmdLine, int iCmdShow)
{
    static char szAppName[] = "ShowBit" ;
    HWND hwnd ;
    MSG msg ;
    WNDCLASSEX wndclass ;

    wndclass.cbSize = sizeof (wndclass) ;
    wndclass.style = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszClassName = szAppName ;
    wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION) ;

    RegisterClassEx (&wndclass) ;

    hwnd = CreateWindow (szAppName, "Show Bitmaps from BITLIB (Press Key)",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT,
        CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

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

    return msg.wParam ;
}

void DrawBitmap (HDC hdc, int xStart, int yStart, HBITMAP hBitmap)
{
    BITMAP bm ;
```

(continued)

```

HDC hMemDC ;
POINT pt ;

hMemDC = CreateCompatibleDC (hdc) ;
SelectObject (hMemDC, hBitmap) ;
GetObject (hBitmap, sizeof (BITMAP), (PSTR) &bm) ;
pt.x = bm.bmWidth ;
pt.y = bm.bmHeight ;

BitBlt (hdc, xStart, yStart, pt.x, pt.y, hMemDC, 0, 0, SRCOPY) ;

DeleteDC (hMemDC) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    static HINSTANCE hLibrary ;
    static int iCurrent = 1 ;
    static HBITMAP hBitmap ;
    HDC hdc ;
    PAINTSTRUCT ps ;

    switch (iMsg)
    {
        case WM_CREATE :
            if ((hLibrary = LoadLibrary ("BITLIB.DLL")) == NULL)
                DestroyWindow (hwnd) ;

            return 0 ;

        case WM_CHAR :
            if (hLibrary)
            {
                iCurrent ++ ;
                InvalidateRect (hwnd, NULL, TRUE) ;
            }

            return 0 ;

        case WM_PAINT :
            hdc = BeginPaint (hwnd, &ps) ;

            if (hLibrary)
            {
                if (NULL == (hBitmap = LoadBitmap (hLibrary,
                    MAKEINTRESOURCE (iCurrent))))
                {
                    iCurrent = 1 ;
                }
            }
    }
}

```

(continued)

```

hBitmap = LoadBitmap (hLibrary,
    MAKEINTRESOURCE (iCurrent)) ;
}

if (hBitmap)
{
    DrawBitmap (hdc, 0, 0, hBitmap) ;
    DeleteObject (hBitmap) ;
}

EndPaint (hwnd, &ps) ;
return 0 ;

case WM_DESTROY :
    if (hLibrary)
        FreeLibrary (hLibrary) ;

    PostQuitMessage (0) ;
    return 0 ;

return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

```

During processing of the WM_CREATE message, the SHOWBIT program gets a handle to BITLIB.DLL:

```

if ((hLibrary = LoadLibrary ("BITLIB.DLL")) == NULL)
    DestroyWindow (hwnd) ;

```

If BITLIB.DLL isn't in the current directory, SHOWBIT will search for it as discussed earlier in this chapter. If it can't find the library at that point, Windows displays a message box noting the problem. When the user presses OK, *LoadLibrary* returns a NULL value, in which case SHOWBIT terminates.

SHOWBIT can obtain a handle to a bitmap by calling *LoadBitmap* with the library handle and the number of the bitmap:

```

hBitmap = LoadBitmap (hLibrary, MAKEINTRESOURCE (iCurrent)) ;

```

This returns an error if the bitmap corresponding to the number *iCurrent* isn't valid or if not enough memory exists to load the bitmap.

While processing the WM_DESTROY message, SHOWBIT frees the library:

```

FreeLibrary (hLibrary) ;

```

When the last instance of SHOWBIT terminates, the reference count of BITLIB.DLL drops to 0 and the memory it occupies is freed.