MICHAEL J. YOUNG

SOFTWARE TOOLS FOR OS/2®

Creating Dynamic Link Libraries

SOFTWARE TOOLS FOR OS/2

SOFTWARE TOOLS FOR OS/2

Creating Dynamic Link Libraries

Michael J. Young

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn Sydney
Singapore Tokyo Madrid San Juan

Library of Congress Cataloging-in-Publication Data

Young, Michael J.
Software tools for OS/2 : creating dynamic link libraries / Michael J. Young.
p. cm.
Bibliography : p.
Includes index.
ISBN 0-201-51787-6
1. OS/2 (Computer operating system) I. Title.
QA76.76.063Y676 1989
005.4'469--dc20 89-34203
CIP

Production Editor: Amorette Pedersen
Cover Design by: Doliber Skeffington Design
Set in 11-point New Century Schoolbook by Benchmark Productions

ABCDEFGHIJ-AL-89
First Printing, July, 1989

To my mother.

TABLE OF CONTENTS

INTRODUCTION **xiii**

An Overview of the Book xv
How to Use the Book xvii
Tools and Requirements xvii

CHAPTER 1
MAKING PROTECTED-MODE PROGRAMS **1**

Developing a Simple Protected-Mode Program 2
Using the OS/2 API 6
DosSleep 7
KbdCharIn 8
DosExit 9
VioWrtTTy 10
Building the Program 14

Adding Multitasking 19
DosCreateThread 25

Adding Interprocess Communication 28
DosSemRequest 34
DosSemClear 35

CHAPTER 2
PRESENTATION MANAGER PROGRAMS **37**

Presentation Manager and Dynamic-Linking 38

The Source Files 39

The C Source File 40
Initialize the Presentation Manager 49
WinBeginPaint 50
WinCreateMsgQueue 50
WinCreateStdWindow 51
WinDefWindowProc 51
WinDestroyMsgQueue 51
WinDestroyWindow 52
WinDispatchMsg 52
WinDrawText 52
WinEndPaint 52
WinGetMsg 53
WinInitialize 53
WinMessageBox 53
WinQueryWindowRect 53
WinRegisterClass 54
WinTerminate 54
Create a Message Queue 55
Register a Window Class 55
Create a Standard Window 55
Process Window Messages 58
Release Presentation Manager Objects 63

The Supporting Files 64
The Resource Script 64
The Header File 65
The Module Definition File 65
The MAKE File 66

CHAPTER 3
HOW DYNAMIC-LINK LIBRARIES WORK 69

The Process 69
Compiling the Program 70
Linking the Program 71
Loading the Program 78
Calling the Dynamic-Link Function 84
Terminating the Program 84

The Uses of Dynamic Linking 86
Dynamic-Link Libraries within the Structure of OS/2 89

CHAPTER 4
CREATING A DYNAMIC-LINK LIBRARY **93**

An Overview of the Process 94

An Example Dynamic-Link Library 97
Writing the C Source Code 109
Writing the Supporting Files 122
Preparing the Dynamic-Link Library 131

Using the Dynamic-Link Library 133

CHAPTER 5
SHARING DATA **141**

Creating an Instance Data Segment 143

Creating a Global Data Segment 148
Virtual Memory 150

Creating Instance and Global Data Segments 155
Using Two Source Files 155
Using a Single Source File 161

Using Instance and Global Data Segments 166

CHAPTER 6
INITIALIZATION AND TERMINATION **175**

Writing Initialization Routines 176
Define the Entry Point 176
Write the C Initialization Function 183
Specify When the Routine is to be Called 184
Testing the Initialization Routine 186

Writing Termination Routines 187
DosExitList 195
Testing the Termination Routine 198

CHAPTER 7
USING THE C RUNTIME LIBRARY **199**

Multiple-Thread Applications 200

Single-Thread Dynamic-Link Libraries 208

Multiple-Thread DLLs and Applications 213
Conclusion 224

CHAPTER 8
USING RUNTIME DYNAMIC LINKING 227

The Basic Steps 228
Step 1 228
DosLoadModule 228
Step 2 230
DosGetProcAddr 231
Step 3 233
Step 4 234
DosFreeModule 235
Advantages of Runtime Dynamic Linking 235
The Disjoint Descriptor Space 236
An Example Application 240

CHAPTER 9
REAL-MODE VERSION OF YOUR LIBRARY 251

Creating Dual-Mode Programs 252
Writing a Real-Mode Version of a DLL 258
Use the OS/2 Family API Functions 269
Observe Real-Mode Restrictions on Family API Functions 269
Do Not Use the C Runtime Library 270
Write Your Code Specifically for Real Mode 270
Differences between Real and Protected Modes 271
Real-Mode Versions 271

CHAPTER 10
ASSEMBLY LANGUAGE DLLS 277

General Guidelines for Assembly Language 278
The Assembly Language Source Code 284
The Module Definition File 290
The Client Program 290
The MAKE File 290
I/O Privileged Dynamic-Link Functions 291

DosPortAccess 302
DosCLIAccess 302
DosR2StackRealloc 304

GLOSSARY **307**

BIBLIOGRAPHY **331**

INTRODUCTION

When OS/2 was introduced in 1987, it was accompanied by a myriad of new concepts and terms. One of the most prominent of these was the expression **dynamic-link library**.

Briefly, a dynamic-link library is a collection of subroutines stored in a disk file, which may be read into memory and called by application programs. The process of loading and accessing a dynamic-link library is known as **dynamic linking**. Traditionally, a subroutine is not stored in a separate disk file, but rather is incorporated directly into the executable file of the program that calls it.

Initially, the concept of dynamic linking may seem esoteric, and the distinction between a dynamic-link function and a normal subroutine may appear academic. However, after reading this book and after working with the operating system, you will realize that dynamic linking is one of the most important features of OS/2, and that dynamic-link libraries have far reaching and practical significance for both software developers and system users. Here are two primary reasons for the unique importance of dynamic linking.

First, the vast collection of services that the operating system provides for application programs (known collectively as the *Application Program Interface*) is implemented as a set of dynamic-link libraries. The dynamic-linking mechanism allows programs written in high-level languages to call these services using the standard function calling protocol. Thus, through dynamic linking, the facilities of the operating system are made readily accessible to programs, and form an integral part of OS/2 applica-

tions. It is important to understand dynamic-linking so that you can make optimal use of these services when writing application programs.

A second reason for the significance of dynamic linking is that you can package your own collections of subroutines as dynamic-link libraries. Whether you are developing a set of functions for use within your own applications, or you are preparing a function module to be sold commercially, a dynamic-link library provides a convenient and efficient vehicle for implementing your module.

The MS-DOS programming world has proven the importance of using packages of prewritten routines for developing application programs. Such function libraries allow the application programmer to focus on the main program logic, and eliminate the need to develop routines for tangential tasks such as managing windows, performing screen I/O, and handling indexed files. In fact, creating and distributing function libraries currently forms the basis for an entire software industry. The added complexity of OS/2 and the Presentation Manager will intensify the need for using prewritten function libraries when developing OS/2 applications.

Function libraries for OS/2 could be packaged in the same manner as those for MS-DOS (typically, either as source code or as binary code within standard object modules). Implementing subroutine packages as dynamic-link libraries, however, offers many advantages. For example, since dynamic-link libraries are stored in separate disk files, executable program files remain small and fast loading. Also, once a dynamic-link library has been read into memory, the code it contains can be shared by several simultaneous application programs; consequently, computer memory is conserved. Furthermore, a program can call a given dynamic-link library regardless of the language in which it is written; thus, dynamic-link libraries form truly generic software tools.

Finally, because functions included in dynamic-link libraries are called in the *same manner* as the basic services of the operating system, dynamic-link libraries can be used to form seamless operating system extensions. In fact, major operating system extensions, such as the Presentation Manager, are implemented as collections of dynamic-link libraries. You can also replace certain OS/2 services with your own

dynamic-link library routines. Specifically, you can replace many of the basic functions for managing the screen, keyboard, and mouse.

As you read this book, you will learn other advantages of dynamic linking, and should come to appreciate the flexibility and elegance of this mechanism.

This book will help you understand and appreciate many of the features of dynamic linking. More importantly, however, it is a practical handbook, written to show you how to *create* dynamic-link libraries. To date, the documentation on writing dynamic-link libraries is sketchy and spread out over many sources. This book gathers this diverse information into a single source, and offers many tips for avoiding problems and optimizing your use of the dynamic-linking mechanism. The book also provides many example listings; most of the discussions begin with concrete programming examples, and subsequently add theoretical and general information to deepen your understanding of the basic techniques.

Note finally that dynamic-link libraries developed according to the techniques given in this book can be used both by basic protected-mode programs and by Presentation Manager applications. Thus, whether you are developing a text-mode OS/2 program or a Presentation Manager application, you will find the programming methods presented in this book relevant to your work.

An Overview of the Book

The treatment of dynamic-link libraries in this book can be divided into three primary areas of emphasis: Chapters 1, 2, and 8 describe how to *use* dynamic-link libraries; Chapter 3 explains how dynamic-link libraries *work*, and the remaining chapters show how to *create* dynamic-link libraries.

Chapter 1 summarizes the techniques for writing a basic OS/2 protected-mode program, and Chapter 2 outlines the methods for developing an elementary Presentation Manager application. These two chapters serve to explain (or review) basic OS/2 programming techniques before the book embarks on the more advanced techniques required to develop dynamic-link libraries. These chapters also show how to call dynamic-link functions from the two primary types of OS/2 programs.

Chapter 3 explains how dynamic-link libraries work, and lays the theoretical groundwork for understanding the techniques presented in the remainder of the book.

Chapter 4 presents the techniques for writing a simple dynamic-link library, and provides a general overview of the entire dynamic-link library development process. Dynamic-linking, however, is a highly flexible mechanism that offers many options and variables. The techniques presented in Chapter 4 use only the simplest of these options; each of the remaining chapters in the book explores one or more of the advanced options.

Chapter 5 describes how to define both shared and non-shared data segments, so that a dynamic-link library can either share data among all programs that call the library, or provide data that is private to each program.

Chapter 6 shows how to create dynamic-link library initialization and termination routines. These routines are especially valuable for dynamic-link libraries that manage resources shared by several programs.

Chapter 7 describes the special versions of the C runtime library that support multiple-thread applications and dynamic-link libraries. It explains the steps that you must take to allow a program or dynamic-link library to use one of these libraries.

Chapter 8 shows how a program can explicitly load a selected dynamic-link library at runtime (normally all referenced dynamic-link libraries are automatically read into memory when the program is loaded).

Chapter 9 summarizes the steps for providing a real-mode version of your dynamic-link library, so that this library can be used by programs designed to run under either real or protected mode, which are known as **dual-mode** programs.

Almost all of the example listings given in chapters 1 through 9 are written in the C language. Chapter 10, however, describes the methods for writing a dynamic-link library in assembly language. This chapter also shows how to use assembly language to write dynamic-link functions that execute with **I/O privilege** (permission to use certain restricted machine instructions).

The Glossary at the end of the book defines many of the technical terms you may encounter while reading this book or other literature on OS/2.

Finally, the Bibliography cites a number of useful books on OS/2 programming, C and assembly language, and the architecture of the 80286 processor.

There are several special uses for dynamic-link libraries that are not covered in this book. For example, these libraries can be used to store OS/2 **resources** (a form of read-only data read stored within an executable file). Also, dynamic-link functions can be used to replace certain OS/2 services (see the documentation on the KbdRegister, MouRegister, and VioRegister OS/2 functions). The basic information presented in this book, however, should make it easy to employ dynamic-link libraries for these and other special uses you may encounter.

How to Use the Book

If you feel the need to review fundamental OS/2 and Presentation Manager programming techniques, you should begin by reading chapters 1 and 2. In all cases, you should read Chapters 3 and 4; these two chapters constitute the heart of the book. Chapter 3 describes the basic mechanisms underlying dynamic linking, and Chapter 4 presents the basic techniques for creating a dynamic-link library.

Once you have finished Chapters 3 and 4, you can read the remaining chapters in any order. Each of these chapters discusses one or more specific features of dynamic-link libraries, and you can select from among them according to your particular needs.

Finally, you should use the comprehensive glossary provided at the end of this book. The literature on OS/2 programming and the C language abounds with technical terms, newly coined expressions, and words used with special meanings. The book employs many of these terms without stopping to define them (or perhaps they are defined only the first time they appear). Accordingly, be sure to use the glossary if a term is unfamiliar, or if you are uncertain of the meaning of a word within a specific context.

Tools and Requirements

Developing OS/2 programs and dynamic-link libraries requires a large number of software tools. The examples in this book were written using

the Microsoft OS/2 Software Development Kit, which was a large (and expensive) collection of software tools supplied as a series of shipments, beginning with pre-release software and culminating with the final retail products. The required tools are now available as the following separate retail products: the Microsoft C compiler version 5.1 (essential), the Microsoft Macro Assembler version 5.1 (most of the examples in the book can be prepared *without* the assembler), the Microsoft Programmer's Toolkit version 1.1 (containing documentation on the OS/2 API functions, and a variety of optional utilities), and IBM OS/2 version 1.1 (the operating system itself, obviously required).

The programming examples in the book are based upon this collection of software tools. Accordingly, it will be easier to use the book if you have these specific tools (or *later* versions of these tools). However, you may be able to develop OS/2 programs and dynamic-link libraries using tools supplied by other vendors; you may also be able to use a high-level language other than C. In either of these cases, you can employ the basic concepts from this book but will need to translate the specific implementation details according to the tools you are using.

Note that the term "programmer's reference," which you will see many times in this book, is a general description referring to either of the following two specific reference books cited in the bibliography: the Microsoft *Programmer's Reference* (supplied with the Programmer's Toolkit) or the IBM *OS/2 Technical Reference*. These reference books fully document each of the operating system functions. The most important OS/2 functions used in this book are described in accompanying figures; however, for additional details on these functions, as well as descriptions of other functions, see either of these two reference books.

Also helpful for writing OS/2 programs are third-party function libraries and programmer's tools. See the Software Offer at the end of the book for a description of several such products I have developed. These products supply complete source code, and are thus valuable as learning resources as well as for facilitating program development.

Finally, this book assumes a working knowledge of C and assembly language, and of OS/2 programming basics. If you need further background knowledge in any of these areas, see the Bibliography for the titles of useful books.

CHAPTER 1

MAKING PROTECTED-MODE PROGRAMS

This chapter explains the basic steps for writing protected-mode programs under OS/2. The applications described here are designed to be run from the OS/2 full-screen command prompt. Although many of these programs can also be run within a window of the Presentation Manager, they cannot use the full set of Presentation Manager features. Chapter 2 describes the basic methods for developing programs specifically for the Presentation Manager—these programs have free access to the facilities of this environment including its extensive set of graphics functions.

Both chapters serve several purposes. First, they provide a brief introduction to (or review of) basic OS/2 programming concepts that are important for understanding the more advanced techniques involved in developing dynamic-link libraries. Next they reveal the general context in which dynamic-link libraries are used. Dynamic-link libraries are not freestanding entities; they are software tools called by applications, and must closely integrate their activities with those of the calling process. Finally, the chapters show how to use dynamic-link libraries. There are few OS/2 applications that do not call dynamic-link library functions—the example programs in these chapters use the dynamic-linking mechanism to access the services of the operating system. (Subsequent chapters will explain how dynamic-link libraries work, how to create your

own dynamic-link libraries to provide custom extensions to the operating system services, and how to package your own collections of routines.)

In this chapter you will discover that you can easily use dynamic-link libraries without knowing how they work or how they are created. From the application programmer's vantage, using a dynamic-link library function is almost identical to calling a subroutine contained in a conventionally linked library; such as the C runtime library. The simplicity of using dynamic-link libraries adds to their importance as software tools; it also makes it possible to postpone the discussion of their inner workings until you have explored several examples of their use.

First you will see how to develop a simple protected-mode application—a variation on the archetypal "hello, world" C program; then add multitasking features to this example; and finally, how to coordinate the activities of the separate program tasks using interprocess communication. Consequently you will learn the fundamental steps to prepare any protected mode application and how to use several of the advanced features of OS/2. The chapter will emphasize the differences between developing protected-mode programs for OS/2 and developing real-mode programs for MS-DOS.

Developing a Simple Protected-Mode Program

Figure 1-1 lists a C source file destined to produce a simple protected-mode OS/2 program. This program repeatedly prints a message on the screen and terminates it when the user presses a key. The message consists of 11 lines form a box containing the string "Hello from main." The program pauses for 1/2 second between messages.

Like most C programs, this example contains calls to external functions (functions not defined within the source file), namely, **DosSleep**, **KbdCharIn**, **DosExit**, and **VioWrtTTY**. As you can see from the source code, these functions perform most of the work of the program. Although the program is written in the same manner as a typical MS-DOS application, there are two important features of these external function calls that are unique to OS/2.

Figure 1-1

```
/*
     Figure 1-1

     You can prepare this program using the following commands:

          cl /c /W2 /G2 /Zp FIG1_1.C
          link /NOI /NOD FIG1_1.OBJ,, NUL, SLIBCE.LIB OS2.LIB, FIG1_6.DEF
*/
#define INCL_DOS
#define INCL_SUB
#include <OS2.H>

void PrintMessage (void);              /* Prints a message on the screen.   */

void main (void)
     {
     KBDKEYINFO Key;                        /* Holds 'KbdCharIn' information*/

     for (;;)
          {
          PrintMessage ();                  /* Print a message on the screen*/

          DosSleep                          /* Create a delay.              */
               (500L);                      /* 1/2 second.                  */
```

```
        KbdCharIn                       /* Test for a key.              */
             (&Key,                     /* Address to receive key info. */
             IO_NOWAIT,                 /* Don't wait for a keystroke.  */
             0);                        /* Keyboard handle.             */

        if (Key.fbStatus)               /* If a key has been typed...   */
             DosExit                    /* Terminate program.           */
                  (EXIT_PROCESS,        /* Terminate entire process.    */
                  0);                   /* Program exit code.           */

        } /* end for loop */

    } /* end main */


void PrintMessage (void)
    {
    VioWrtTTY("*************************************************\r\n",53,0);
    VioWrtTTY("*                                               *\r\n",53,0);
    VioWrtTTY("*                                               *\r\n",53,0);
    VioWrtTTY("*                                               *\r\n",53,0);
    VioWrtTTY("*                                               *\r\n",53,0);
    VioWrtTTY("*                Hello from main.               *\r\n",53,0);
    VioWrtTTY("*                                               *\r\n",53,0);
    VioWrtTTY("*                                               *\r\n",53,0);
    VioWrtTTY("*                                               *\r\n",53,0);
    VioWrtTTY("*                                               *\r\n",53,0);
    VioWrtTTY("*************************************************\r\n",53,0);

    } /* end PrintMessage */
```

These function calls *directly invoke the services of the operating system.* Like MS-DOS, OS/2 provides a collection of services that support application programs; these services are known collectively as the application program interface (API). Under MS-DOS, system services are accessed by generating software interrupts (by means of the INT machine instruction), and values are exchanged between the program and the system through machine registers. Because the C language does not provide an interrupt instruction and because C programs do not have direct access to machine registers, these programs must access the operating system services indirectly through library functions such as **intdos** and **int86**. Under MS-DOS, the mechanism for invoking the operating system is indirect and awkward, and is different from the normal protocol for calling external functions. Consequently, many C programs for MS-DOS are written without direct calls to the system and rely upon the C runtime library routines for obtaining required services, or they include their own low-level routines for accessing the hardware.

Under OS/2, C programs can directly call operating system services in the *same manner* that they call other external functions, such as routines belonging to the C runtime library. Values are passed to the function through a normal list of parameters, the program issues a direct **far** call to the operating system code in memory, and the function directly returns a value to the calling program. System services are simple to access, and the set of OS/2 functions is more comprehensive and in many cases more efficient than services offered by MS-DOS. For C programs written under OS/2, API services provide an important extension to the C runtime library. You can use the OS/2 API to obtain services not provided by the C library; you can also call API services that are equivalent to C library functions to achieve greater efficiency. Since many C library services ultimately call an API function, you can eliminate a function call and the associated overhead by calling the API function instead of the library function. For example, the C library routine **write** calls the OS/2 function **DosWrite**; you can enhance efficiency by calling DosWrite directly.

The second unique feature of the external functions called in Figure 1-1's example program is that they do not reside in normal object or library files. They are contained in one or more **dynamic-link library files**, which are provided with the operating system and are copied to a special

directory when you install the system. The linker doesn't insert the code for these functions into the executable program file; rather, the function code remains in the separate dynamic-link library files until the program is run. When the program is loaded, the dynamic-link library files are also loaded so the program can call the functions they contain.

Using the OS/2 API

The API functions used in the example program—**DosSleep**, **KbdCharIn**, **DosExit**, and **VioWrtTTY**—are summarized in Figures 1-2 through 1-5. All of the OS/2 API functions are called through the dynamic-link mechanism. OS/2 provides four basic categories of functions (in addition to the vast collection of functions available to Presentation Manager applications, which are described in Chapter 2). The first three letters of a function's name indicates its category; these prefixes and the corresponding categories are shown in Table 1-1:

Table 1-1: Function Categories

PREFIX	FUNCTION CATEGORY
Dos	General purpose system functions
Kbd	Keyboard management functions
Mou	Mouse functions
Vio	Screen management functions

The general purpose **Dos** functions provide a variety of services. These functions can be divided into the following general classifications according to the types of services they provide:

- Program startup information
- Memory management
- Disk file and character device I/O
- Disk, directory, and file management

- Low-level device control
- Multitasking of threads
- Multitasking of processes
- Multitasking of screen groups
- Interprocess communication
- Time and date management
- National language support
- Runtime dynamic linking
- Device monitor management
- Error handling
- Additional miscellaneous functions

Figure 1-2: The DosSleep OS/2 function.

DosSleep

☐ *Purpose*: Suspends the execution of the calling program thread for a specified time period.

☐ *Prototype*: USHORT APIENTRY DosSleep

(ULONG ulTime);	Period of time to suspend execution in milliseconds; the time is rounded up to the nearest clock tick; a value of 0 causes the thread to yield the remainder of its time slice.

☐ *Return Value*: If successful, the function returns zero. If an error occurs, it returns a nonzero error code. The following is the possible error code:

ERROR_TS_WAKEUP	The function has returned before the specified time has elapsed.

Figure 1-3: The KbdCharIn OS/2 function.

KbdCharIn

☐ *Purpose*: Reads a keystroke from the keyboard.

☐ *Prototype*: USHORT APIENTRY KbdCharIn

(PKBDKEYINFO pkbciKeyInfo;	Points to a KBDKEYINFO structure that receives information on the keystroke—see the definition of this structure below.
USHORT fNoWait,	A flag specifying whether the function should wait for a keystroke; it can be assigned one of the following values: IO_WAIT: Wait for a key to be entered if one is not available in the system buffer. IO_NOWAIT: Return immediately whether or not a keystroke is available; if a key was read, the **fbStatus** field of the KBDKEYINFO structure is given a nonzero value.
HKBD hkbd);	The keyboard handle; you can pass the value 0 or the handle of a logical keyboard supplied by **KbdOpen**.

☐ *Structure*:

```
typedef struct _KBDKEYINFO
        {
        UCHAR           chChar;          /* The character code.*/
        UCHAR           chScan;          /* The scan code.*/
        UCHAR           fbStatus;        /* Status: nonzero if a key was read. */
        UCHAR           bNlsShift;       /* A reserved field.*/
        USHORT fsState;                  /* State of the shift keys.*/
        ULONG           time;            /* Time when keystroke was entered.*/
}
KBDKEYINFO;
```

☐ *Return Value*:	If successful, the function returns zero. If an error occurs, it returns a nonzero error code. The following is the possible error code:
ERROR_KBD_INVALID_IOWAIT	The value you passed in the **fNoWait** parameter is invalid.

Figure 1-4: The DosExit OS/2 function.

DosExit

☐ *Purpose*: Terminates a thread or a process.

☐ *Prototype*: VOID APIENTRY DosExit

(USHORT fTerminate,	A flag indicating whether to terminate a single thread or the entire process; it may be assigned one of the following values: EXIT_THREAD: Terminate only the current thread. EXIT_PROCESS: Terminate the entire process.
USHORT usExitCode);	The program exit code.

The **Kbd**, **Mou**, and **Vio** sets of functions are known as **subsystems**. Each of these subsystems provides a comprehensive set of services for managing a specific device.

When you call an OS/2 API function from a C program, you must include the corresponding function declaration in your programs; otherwise, the compiler will generate the wrong function calling conventions, and may fail to properly convert parameter types. All required function declarations, type definitions, and constant definitions are supplied in a set of header files provided with the Microsoft C compiler (version 5.1 or later) and with the Microsoft OS/2 Programmer's Toolkit. You need explicitly

include only the single header file OS2.H, which contains include statements for the other header files. To minimize the amount of compile time, however, only the most commonly used declarations and definitions are included by default.To force the inclusion of additional header information, you must define certain symbolic constants before including OS2.H. For example, the program of Figure 1-1 begins with the following preprocessor commands:

```
#define INCL_DOS
#define INCL_SUB
#include <OS2.H>
```

Figure 1-5: The VioWrtTTY OS/2 function.

VioWrtTTy

☐ *Purpose*: Writes a character string to the screen at the current cursor position.

☐ *Prototype*: USHORT APIENTRY VioWrtTTY

(PCH pchString,	Address of the string to write.
USHORT cbString,	Length of the string.
HVIO hvio);	For most programs, this parameter should be set to 0; for programs that call the **VioCreatePS** function (advanced VIO programs), this parameter should supply the video handle obtained from **VioCreatePS**.
☐ *Return Value*:	If successful, the function returns 0. If an error ocurrs, it returns a nonzero error code. The following is the possible error code:
	ERROE_VIO_INVALID_HANDLE: The value passed in the **hvio** parameter is invalid.

Defining the symbolic constant INCL_DOS causes the inclusion of all header information for the Dos system functions, and INCL_SUB causes inclusion of the header information for the Kbd, Mou, and Vio subsystem functions. You can define other constants to include larger or smaller subsets of information. For a complete list and an explanation of these constants, see the compiler documentation or the OS/2 Programmer's Toolkit manual

In addition to the required function declarations, the header files contain useful type and symbolic constant definitions. For example, the first parameter to the function KbdCharIn is declared as a pointer to a KBDKEYINFO structure. Since this structure type is defined in the OS/2 header files, you need merely declare a variable that is an instance of this type to receive keyboard information, such as the variable *Key* declared in Figure 1-1 as follows:

```
KBDKEYINFO Key;
```

See Figure 1-3 for the definition of the KBDKEYINFO structure. Another example of a variable type defined by the OS/2 header files is USHORT, which is defined as follows (in OS2DEF.H):

```
typedef unsigned short USHORT;
```

As you can see in Figures 1-2 through 1-5, USHORT is the return type for most of the API functions, except those that do not return a value, such as DosExit. The API return values communicate the error status, which will be explained shortly. When you declare a variable that interfaces with an API function (that is, one that receives a return value or is passed as a parameter), you should use the uppercase type names provided in the OS/2 header files whenever possible rather than the basic C types, since these basic types are subject to change. For example, if you declare a variable to receive an API function return code, you should declare it as type USHORT, rather than **unsigned short**. In the initial OS/2 header files, USHORT was defined as **unsigned int**. Not only might the basic C types be adjusted as the programming tools evolve, but these types may be changed if OS/2 were ported to a computer that was based on another processor.

The program in Figure 1-1 uses several of the symbolic constant definitions provided in the OS/2 header files, namely IO_NOWAIT, passed to KbdCharIn, and EXIT_PROCESS, passed to DosExit. Using these definitions rather than raw numeric values makes the program more understandable. Also, the function explanations in the OS/2 programmer's reference refer to the symbolic constants rather than the numeric values.

Finally, the OS/2 header files provide a collection of useful macro definitions. Using macros is especially valuable for Presentation Manager programming—see examples in later chapters.

Almost all of the OS/2 API functions in the categories listed at the beginning of this section, return an error status code to the calling program. The only exceptions are the functions that do not return values, which are declared with the return type VOID. If successful, all of these functions return zero and, if an error does occur, they return a nonzero code indicating the specific error. The documentation for each function (in the OS/2 programmer's reference) lists the possible error return codes that can be returned by that function. These codes are referred to by symbolic constants, all of which are defined in the OS/2 header files. For example, the function **DosRead**, which is the basic OS/2 function for reading a file or device, can return one of four error codes. The symbolic constants for these errors (as listed in the programmer's reference) are as follows:

```
ERROR_ACCESS_DENIED
ERROR_BROKEN_PIPE
ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION
```

You can include the definitions of the symbolic constants for error codes in your program by defining INCL_DOSERROR before including OS2.H. For more information on the API functions, see the OS/2 programmer's reference, *Programmer's Guide to OS/2*, or one of the other books on OS/2 kernel programming cited in the Bibliography.

You can see that the set of functions available under OS/2 is usually more complete than that under MS-DOS. A comprehensive and efficient API is an important requirement in a multitasking operating system. Whereas MS-DOS programs can efficiently perform a wide variety of tasks by circumventing the operating system and directly accessing the underlying hardware, OS/2 programs usually must obtain desired services through the operating system. In the interests of protecting the integrity of the system, OS/2 protected-mode programs are subject to several restrictions. These restrictions are summarized as follows:

- OS/2 programs cannot invoke the routines provided by the BIOS of IBM-compatible microcomputers. (The BIOS routines are accessed through software interrupts; however, OS/2 programs are not allowed to issue software interrupts. Also, the BIOS code in AT-class machines cannot run in protected mode.) MS-DOS programs typically rely upon the BIOS for obtaining a wide variety of services for controlling devices. The OS/2 API subsystems, however, provide many of these same services. For instance, the Vio subsystem provides all of the services that MS-DOS programs normally obtain through the BIOS (via interrupt 10h).

- OS/2 programs cannot access arbitrary memory locations. OS/2 takes advantage of the hardware protection provided by the 80286 processor to prevent programs from accessing memory addresses in segments that have not been explicitly allocated to the process or designated as globally accessible. Many MS-DOS programs, for example, achieve fast video output by writing directly to the video refresh buffer in high memory. Although OS/2 programs cannot normally write directly to video memory, the Vio functions provide nearly comparable performance.

- OS/2 programs usually cannot write directly to I/O port addresses to control hardware devices. Exceptions are the I/O-privileged routines discussed in Chapter 10). However, OS/2 provides a large family of commands for directly controlling devices—the I/O control commands accessed through the **DosDevIOCtl** API function. See the OS/2 programmer's reference for more information.

The example program of Figure 1-1, written to illustrate the use of the OS/2 API services, did not call any C library functions. However, OS/2

protected-mode programs are free to use the routines of the C library, since special versions of the library are provided that have been written specifically for the protected mode—these library versions are supplied with Microsoft C version 5.1 and later. Accordingly, the MS-DOS and OS/2 versions of programs that obtain most of their services through the C library are very similar. You might be able to port an MS-DOS program that uses the C library and does not contain low-level code by simply relinking the program with the protected-mode version of the library.

However, you cannot use the standard protected-mode C library when writing dynamic-link functions—there are severe restraints on using this library for programs that employ multitasking. The Microsoft C compiler now provides special versions of the protected-mode library that eliminate these restrictions. The use of these library versions is discussed in Chapter 7.

Building the Program

The example program of Figure 1-1 can be prepared through the following two commands:

```
cl /c /W2 /G2 /Zp FIG1_1.C
```

and:

```
link /NOI /NOD FIG1_1.OBJ,, NUL, SLIBCE.LIB OS2.LIB, FIG1_6.DEF
```

The first command compiles the program and produces an object file (FIG1_1.OBJ); the second command links the object file with the required library code and produces a protected-mode executable program (FIG1_1.EXE). This program can be run only in protected mode. If you attempt to run it under MS-DOS or in the DOS-compatibility environment of OS/2, it will display an error message and immediately terminate.

In general, compiling a protected-mode program is the same as compiling a real-mode program—it generates a standard object file. The following are the recommended command line switches for compiling the example program of Figure 1-1:

- **/c** This flag tells the compiler to generate an object file without running the linker (the default action of the CL command is to compile and link the program). To gain greater control over the compiling and linking processes, these two steps are performed separately.

- **/W2** This flag generates a higher level of warnings than are emitted by default. For example, it will warn you if you call a function that has not been declared previously in the source file.

- **/G2** This flag allows the compiler to generate instructions that are unique to the 80286 and later model processors. Since a protected-mode program must run on at least an 80286 processor, it makes sense to take advantage of the advanced instructions provided by this processor.

- **/Zp** This flag forces the compiler to pack all structures, meaning that each field of a structure is located at the first available byte address. By default, the C compiler aligns all structure members except **char** and **unsigned** char on word boundaries. In general, packing structures makes it simpler to exchange data with assembly language routines and with certain routines belonging to the OS/2 API.

Most of the differences between building a protected-mode program and building a real-mode program appear in the linking step. Although the compiler generates a standard object file for both protected and real mode programs, the linker must produce a special EXE file for protected mode programs. Specifically, protected-mode executable programs have a unique file header, which will be described in Chapter 3. The example LINK command given in this section illustrates several important new requirements for producing a protected-mode program.

First, you must use a version of the linker designed to generate protected-mode programs such as the linker supplied with OS/2, or with the Microsoft C compiler version 5.1 or later. The following are the recommended command line switches used for linking the example program:

- **/NOI** (NOIGNORECASE) This flag tells the linker to distinguish between uppercase and lowercase letters in names. This op-

tion is useful with a case-sensitive language such as C, if you use identifiers that differ only by the case of one or more letters.

- **/NOD** (NODEFAULTLIBRARYSEARCH) This flag tells the linker not to search any default libraries that may have been specified in the object file. The C compiler normally writes the names of one or more default search libraries into the object file. Since the names of your library files may not match the default names, the best technique is to include the /NOI switch and then fully specify all library names in the linker command line.

A second requirement for linking a protected-mode program is to specify the name of the protected-mode version of the C runtime library for the current memory model. The example LINK command designates the library name SLIBCE.LIB—the name of the small-model, protected-mode version of the C library in the system used to develop the example programs in this book. If you have given this library file a different name, be sure to substitute your name (for example, one of the names recommended in the compiler documentation is SLIBCEP.LIB).

Although the example program contains no direct calls to functions in the C library, the linker must obtain the C startup code from this library. You must link the program with the appropriate C library to resolve the references to the C startup function generated by the compiler. Note that the C runtime library is a conventional library file that is linked using the conventional linking technique; the complete body of code and data belonging to all referenced object modules is inserted into the final executable file.

The third new requirement illustrated by the example LINK command, is that you must specify another library file to resolve all references to the OS/2 API calls. Depending upon the version of the operating system or the development tools, this library file is named either OS2.LIB (as in the example command), or DOSCALLS.LIB—use the correct name for your installation.

OS2.LIB (or DOSCALLS.LIB) is not a conventional library file containing object code; rather, it is a special library for resolving references to dynamic-link functions known as an **import library**. An import library generally does not contain program code or data. Instead, for each dynamic-link function, it provides a record giving the name of the

dynamic-link library containing the function and the entry point of the function within this library. In other words, rather than supplying the actual object code, it supplies all the information needed to load and call the function when the program is run.

When the linker resolves a reference to a dynamic-link function, instead of copying code and data into the executable file, it merely copies an import record for this function (listing the dynamic-link library name and the function entry point) into the program header. When the program is run, the system knows where to find the actual code and data for each dynamic-link function called by your program. Also, when the program is run, all required dynamic-link libraries are loaded into memory, and each call to a dynamic-link function within the program code is supplied with the far address of the function in memory.

The dynamic-link libraries containing the actual code and data for the API functions are stored in an appropriate directory on the hard disk (they are copied into this directory when you install the operating system). These files all have the .DLL extension. The application programmer, however, does not need to know the names of the files, since the names are contained in the import library and the linker automatically inserts them into the program header. Note that OS2.H (or DOSCALLS.H) is provided with the operating system. In general, a supplier of dynamic-link libraries also furnishes the corresponding import library to allow programs to call the dynamic-link functions. Chapter 4 will explain an alternative method.

The C runtime library and the OS/2 import file must be either in the current directory or in a directory specified by the LIB environment variable at the time you link your program—see your compiler documentation for an explanation of the LIB variable. Also, the required dynamic-link library files must be in a directory specified by the LIBPATH configuration command. See your OS/2 user's guide for more information of this command.

Chapter 3 explains the dynamic-link mechanism in greater detail, and Chapter 4 shows how to create both dynamic-link libraries and the corresponding import libraries.

The fourth new feature exhibited by the example LINK command is the use of a **module definition file**—the name is specified as the last

item on the linker command line. You can provide a module definition file when using any version of the linker that supports Microsoft Windows or OS/2 applications; this file can be used to specify many of the attributes of the executable program. Providing a module definition file is required when linking Presentation Manager applications and dynamic-link libraries. For other types of protected-mode programs, a module definition file is optional; if it is omitted, the linker uses the default values for all items that can be specified through this file. The name of the module definition file used for linking the example program of Figure 1-1 is FIG1_6.DEF. This file is listed in Figure 1-6, and contains the following two commands:

```
NAME FIG1_1 WINDOWCOMPAT
```

and:

```
PROTMODE
```

Figure 1-6

```
;    Figure 1-6
;    A module definition file for the program of Figure 1-1


NAME FIG1_1 WINDOWCOMPAT


PROTMODE
```

The presence of a NAME command indicates that the output file is to be an executable program rather than a dynamic-link library. This command is followed by the name of the executable file, FIG1_1, and a flag indicating the type of the application, WINDOWCOMPAT, which signifies that the program can run within a window of the Presentation Manager. The WINDOWCOMPAT flag does not mean that the program is a Presentation Manager application, merely that it can be run safely within a window managed by the Presentation Manager. If you do not specify the

application type, the system will automatically run the program in a separate screen group. For an explanation of screen groups see the end of the section on Adding Multitasking.

The PROTMODE command indicates that the application can be run only in protected-mode, and allows the linker to omit certain items of information from the program header. Figure 1-7 summarizes the OS/2 linking process and illustrates the differences between the conventionally linked C library code and the dynamically linked OS/2 API function code.

Adding Multitasking

Before concluding this chapter on developing protected-mode programs, these last two sections briefly introduce two of the advanced features offered by OS/2: **multitasking** and **interprocess communication**.

The example program in Figure 1-1, like programs written for MS-DOS, consists of a single thread. A **thread** is the execution of a series of instructions within a program. In a single-thread application only one sequence of machine instructions is executed at a given time; therefore, such a program performs only one task at a time. The example program in Figure 1-8 is similar to the one in Figure 1-1; however, this version of the program starts two additional threads of execution. Accordingly, it performs three simultaneous, independent tasks, namely, three separate portions of this program run at the same time. You can compile and link the program in the manner described in the last section—Figure 1-9 provides a module definition file for linking this program. You can prepare this program using the following two commands:

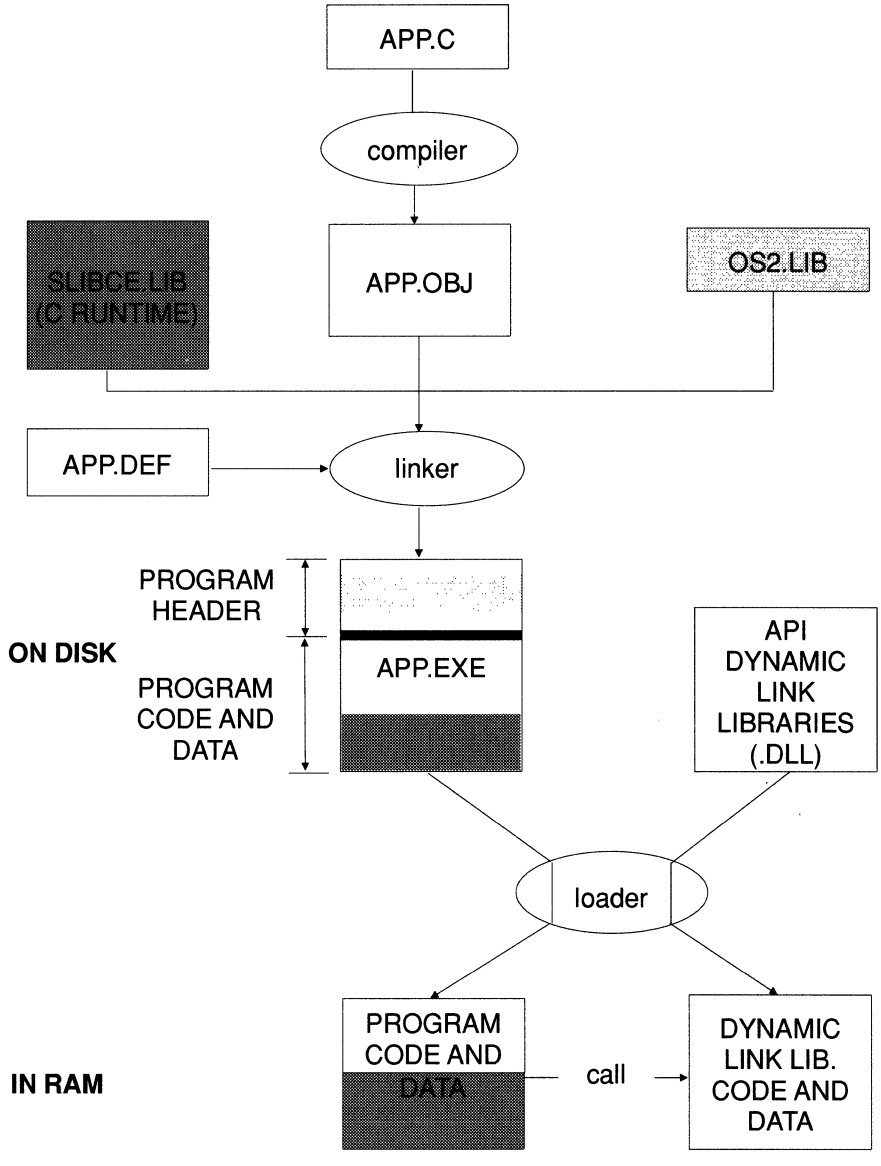
```
cl  /c  /W2  /ASw  /G2  /Zp
```

and:

```
link  /NOI  /NOD  NUL,  SLIBCE.LIB  OS2.LIB,
```

Note that these commands use the same switches as those used to prepare the program of Figure 1-1, except for the additional compiler switch, **/ASw**. This switch is required because the program runs more than one simultaneous thread.

Figure 1-7: The OS/2 linking process.



The **S** option specifies the small memory model (although a C program is compiled under this model by default, when you employ the **/A** flag, you must explicitly indicate the desired memory model). The **w** option tells the compiler that the contents of the stack segment register (SS) are not equal to the contents of the data segment register (DS). In a normal C program, these two registers are equal (the default C Stack is located in the same segment as the program data), and by default the compiler generates code that assumes this equality. However, additional threads started by the program do not use the default C stack; rather, each thread is assigned its own stack, which can be located within another segment.

The program is linked in the same way as Figure 1-1's program. Figure 1-9 provides a module definition file for performing the linking step.

Figure 1-8

```
/*
     Figure 1-8

     You can prepare this program using the following commands:

         cl /c /W2 /G2 /Zp FIG1_8.C
         link /NOI /NOD FIG1_8.OBJ,, NUL, SLIBCE.LIB OS2.LIB, FIG1_9.DEF
*/
#define INCL_DOS
#define INCL_SUB
#include <OS2.H>

#pragma check_stack (off)          /* Disable stack probes.                */

char StackT2 [3072];               /* Reserve a 3K stack for thread 2.     */
```

```
char StackT3 [3072];                /* Reserve a 3K stack for thread 3.       */
void far Thread2 (void);            /* Function executed by second thread.    */
void far Thread3 (void);            /* Function executed by third thread.     */

void PrintMessage (int Thread);     /* Prints a message on the screen.        */

void main (void)
     {
     TID ThreadID;                  /* Receives ID of new thread.             */
     KBDKEYINFO Key;                /* Holds 'KbdCharIn' information.         */

     DosCreateThread                /* Start second thread.                   */
          (Thread2,                 /* Address of function executed by thread 2.  */
          &ThreadID,                /* Address to receive new thread ID.      */
          StackT2 + sizeof (StackT2));          /* Address of top of stack. */

     DosCreateThread                /* Start third thread.                    */
          (Thread3,                 /* Address of function executed by thread 3.  */
          &ThreadID,                /* Address to receive new thread ID.      */
          StackT3 + sizeof (StackT3));          /* Address of top of stack. */

     KbdCharIn                      /* Pause until a key is pressed.          */
          (&Key,                    /* Address to receive key information.    */
          IO_WAIT,                  /* Wait for a keystroke.                  */
          0);                       /* Keyboard handle.                       */

     DosExit                        /* Terminate program.                     */
          (EXIT_PROCESS,            /* Terminate ALL threads.                 */
```

```
        0);                 /* Program exit code.                          */
    } /* end main */


void far Thread2 (void)     /* Executed by second thread.                  */
    {
    for (;;)
        {
        PrintMessage (2);   /* Print a message on the screen.              */
        DosSleep            /* Create a delay.                             */
            (500L);         /* 1/2 second.                                 */
        }

    } /* end Thread2 */


void far Thread3 (void)     /* Executed by third thread.                   */
    {
    for (;;)
        {
        PrintMessage (3);   /* Print a message on the screen.              */
        DosSleep            /* Create a delay.                             */
            (500L);         /* 1/2 second.                                 */
        }

    } /* end Thread3 */


void PrintMessage (int Thread)
    {
    VioWrtTTY("**************************************************\r\n",53,0);
    VioWrtTTY("*                                                *\r\n",53,0);
    VioWrtTTY("*                                                *\r\n",53,0);
```

```
VioWrtTTY("*                                                    *\r\n",53,0);
VioWrtTTY("*                                                    *\r\n",53,0);
if (Thread == 2)
VioWrtTTY("*              Hello from Thread 2                   *\r\n",53,0);
else
VioWrtTTY("*              Hello from Thread 3                   *\r\n",53,0);
VioWrtTTY("*                                                    *\r\n",53,0);
VioWrtTTY("*                                                    *\r\n",53,0);
VioWrtTTY("*                                                    *\r\n",53,0);
VioWrtTTY("*                                                    *\r\n",53,0);
VioWrtTTY("**************************************************\r\n",53,0);

} /* end PrintMessage */
```

Figure 1-9

```
;               Figure 1-9
;               A module definition file for the program of Figure 1-8

NAME FIG1_8 WINDOWCOMPAT

PROTMODE
```

When the new version of the example program begins running, it consists (like all protected-mode applications) of a single thread—known as thread number 1. The program then makes two calls to the API function **DosCreateThread** (explained in Figure 1-10) to start two additional threads of execution (known as threads 2 and 3). Each of these two new threads executes a function that repeatedly prints a message box on the screen, which includes the number of the thread displaying the message. After starting these threads, thread number 1 waits for a keystroke, when

the user presses a key, this thread terminates the program (it terminates all three threads by calling **DosExit** and passing the value EXIT_PROCESS as the first parameter).

Figure 1-10: The DosCreateThread OS/2 function.

DosCreateThread

☐ *Purpose*: Begins a new thread of execution.

☐ *Prototype*: USHORT APIENTRY DosCreateThread

(VOID (FAR *) pfnFunction (VOID),	Address of the function to be executed by the new thread.
PTID ptidThread,	Address of variable to receive the identifier of the new thread.
PBYTE pbThrdStack)	Address of the top of the stack for the new thread.

☐ *Return Value*: If successful, the function returns zero. If an error occurs, it returns a nonzero error code. The following are the possible error codes:

ERROR_NO_PROC_SLOTS	No more threads can be started.
ERROR_NOT_ENOUGH_MEMORY	Insufficient free memory.

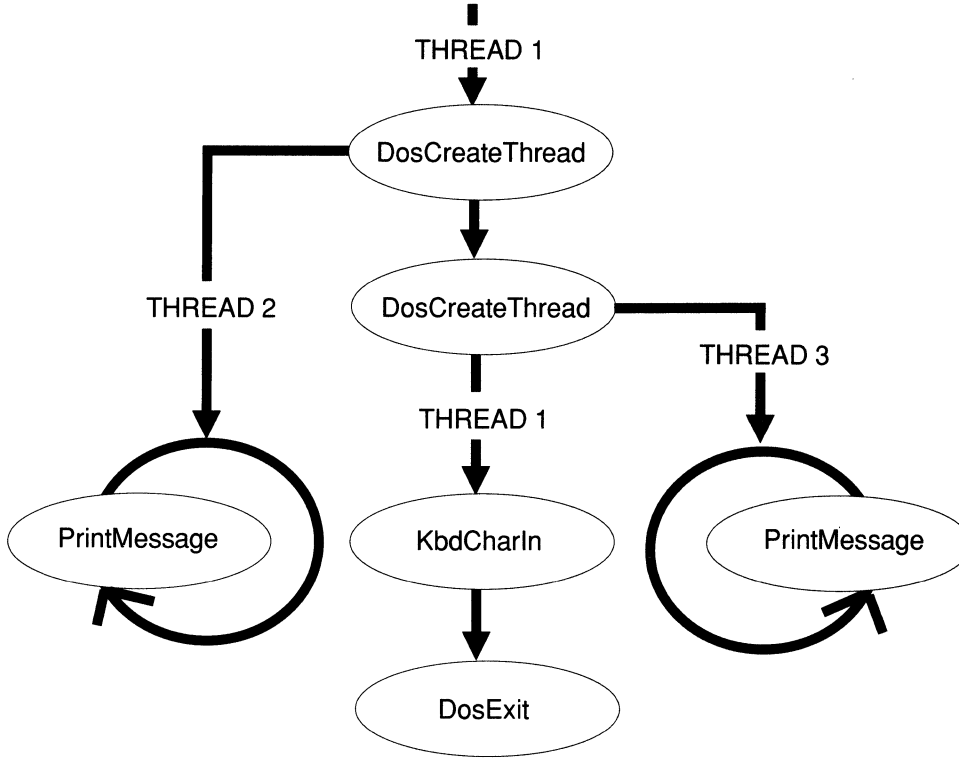
Figure 1-11 illustrates the activities of the three threads.

The initial program thread uses the default C stack, while each new thread you start has its own private stack. When you initiate a new thread, you must reserve an area in memory to serve as the thread's stack. You can simply define an array of the required size within the program's data segment, or you can dynamically allocate a block of memory from the system. The address of the *top* of this stack—that is, the first byte beyond the allocated block—is supplied to **DosCreateThread** as the third parameter. The technical documentation recommends that if a thread calls an API function, it should have at least 2 kilobytes of free

stack space in addition to space for its own needs. Accordingly, the example program allocates a 3-kilobyte stack for each new thread.

You must disable stack checking for any function executed by a new thread, since the stack-checking routine assumes that the function uses the default C program stack, and will immediately abort the program if the function uses a different stack area.

Figure 1-11: The threads in the example program.



Stack checking can be disabled for the entire program through the **/Gs** command line option, or for selected functions through the **check_stack** pragma—as in the example program.

In general, you cannot call standard C library functions from a multiple-thread program. These functions not only contain calls to the stack-checking routine, but most of them are not designed to be called by more than one program thread at a given time—the functions are said to be **nonreentrant**. Chapter 7 describes how to use alternative versions of the C runtime library that fully support multiple-thread applications.

IThe processor can execute only a single thread at a given instant. OS/2, however, switches the processor from thread to thread so rapidly that the threads appear to be running simultaneously. The operating system scheduler runs each thread in turn for a small period of time known as a **time slice**. Since the processor apportions time slices among individual threads, a thread can also be defined as the **basic dispatchable entity** under OS/2.

In the multitasking example program of Figure 1-8, threads 2 and 3 attempt to draw message boxes on the screen. Since these threads run concurrently, however, the video output generated by the threads becomes interspersed. If you run this program, you will not see separate message boxes from each thread, but rather a confusing mixture of lines drawn by both threads. Clearly, the program requires some way of coordinating the activities of threads 2 and 3, so that while one thread is drawing a message box, the other thread is temporarily prevented from writing to the screen until the box is completed. The next section describes how to achieve this coordination.

Before leaving the discussion of multitasking, it is important to define the concept of a **process**, and to distinguish a process from a thread. A process is an instance of the execution of a program. A program is a body of code and data residing in a disk file or in memory; when you begin executing a program, you create a process. A given process consists of one or more individual threads. For example, running the first example program in this chapter creates a process consisting of a single thread, and running the second program creates a process that comprises three threads.

Under OS/2, a process is also the entity that owns objects such as memory segments (code and data), file handles, and connections to dynamic-link libraries. The threads that constitute a process *share* the code and data segments and all other objects owned by the process. Also, as you will see later in the book, a dynamic-link library function runs as part of the process that calls it, and therefore has complete access to all objects belonging to that process.

As stated previously, threads of execution run concurrently. Furthermore, these concurrent threads are not confined to a single process; rather, the scheduler runs all active threads in the system regardless of the processes that own them. The scheduler does not know of the existence of processes; it sees only a collection of threads. Consequently, under OS/2 you can run not only multiple threads of execution within a single process, but also multiple simultaneous processes.

Under OS/2, the collection of processes that are running simultaneously can belong to more than one **screen group**. A screen group, also known as a **session**, is a collection of one or more processes that share the screen, keyboard, and mouse. When you run OS/2, you can switch among screen groups (using the Alt-Esc system hot key or the Task Manager utility), bringing one at a time into the foreground. Although you see and control only the foreground screen group, processes in the background screen groups continue to run. Note that the Presentation Manager and all the applications running within its windows belong to a single screen group.

Adding Interprocess Communication

The final version of the example protected-mode program, which is listed in Figure 1-12, uses a semaphore to coordinate the activities of the two threads that write to the screen. Figure 1-13 provides the module definition file for linking the program. You can prepare this program using the following two commands:

```
cl  /c  /W2  /ASw  /G2  /Zp FIG1_12.c
```

and:

```
link /NOI /NOD FIG1_12.OBJ,,NUL, SLIBCE.LIB OS2.LIB, FIG1_13.DEF
```

A semaphore is a software flag used to synchronize the activities of two or more program threads by exchanging simple "stop" and "go" information. A semaphore is normally in one of two states: **set** or **clear**. A set semaphore generally indicates that the thread should stop and wait for the semaphore to be cleared by another thread, and a clear semaphore generally means that the thread can continue.

Figure 1-12

```
/*
     Figure 1-12

     You can prepare this program using the following commands:

         cl /c /W2 /G2 /Zp FIG1_12.C
         link /NOI /NOD FIG1_12.OBJ,, NUL, SLIBCE.LIB OS2.LIB, FIG1_13.DEF
*/
#define INCL_DOS
#define INCL_SUB
#include <OS2.H>

#pragma check_stack (off)          /* Disable stack probes.                */

char StackT2 [3072];               /* Reserve a 3K stack for thread 2.     */
char StackT3 [3072];               /* Reserve a 3K stack for thread 3.     */

void far Thread2 (void);           /* Function executed by second thread.  */
void far Thread3 (void);           /* Function executed by third thread.   */
```

```
ULONG Sem = 0;                      /* Semaphore for synchronizing threads.   */
void PrintMessage (int Thread);     /* Prints a message on the screen.        */

void main (void)
     {
     TID ThreadID;              /* Receives ID of new thread.                 */
     KBDKEYINFO Key;            /* Holds 'KbdCharIn' information.             */

     DosCreateThread            /* Start second thread.                       */
          (Thread2,             /* Address of function executed by thread 2.  */
          &ThreadID,            /* Address to receive new thread ID.          */
          StackT2 + sizeof (StackT2));          /* Address of top of stack. */

     DosCreateThread            /* Start third thread.                        */
          (Thread3,             /* Address of function executed by thread 3.  */
          &ThreadID,            /* Address to receive new thread ID.          */
          StackT3 + sizeof (StackT3));          /* Address of top of stack. */

     KbdCharIn                  /* Pause until a key is pressed.              */
          (&Key,                /* Address to receive key information.        */
          IO_WAIT,              /* Wait for a keystroke.                      */
          0);                   /* Keyboard handle.                           */

     DosExit                    /* Terminate program.                         */
          (EXIT_PROCESS,        /* Terminate entire process.                  */
          0);                   /* Program exit code.                         */

     } /* end main */
```

```
void far Thread2 (void)        /* Executed by thread 2.                    */
    {
    for (;;)
         {
         DosSemRequest         /* Wait for semaphore to clear and then set it.*/
              (&Sem,           /* Semaphore handle (its far address).      */
              -1L);            /* Wait flag:  -1 means wait forever.       */

         DosSleep (500L);      /* Create a 1/2 second pause.               */

         PrintMessage (2);     /* Print a message on the screen.           */

         DosSemClear           /* Clear the semaphore.                     */
              (&Sem);          /* Semaphore handle.                        */

         DosSleep (0L);        /* Yield remainder of time slice.           */

         }

    } /* end Thread2 */


void far Thread3 (void)        /* Executed by thread 3.                    */
    {
    for (;;)
         {
         DosSemRequest         /* Wait for semaphore to clear and then set it.*/
              (&Sem,           /* Semaphore handle (its far address).      */
```

```
            -1L);          /* Wait flag:  -1 means wait forever.          */
        DosSleep (500L);   /* Create a 1/2 second pause.                  */

        PrintMessage (3);  /* Print a message on the screen.              */

        DosSemClear        /* Clear the semaphore.                        */
            (&Sem);        /* Semaphore handle.                           */

        DosSleep (0L);     /* Yield remainder of time slice.              */

        }

    } /* end Thread3 */

void PrintMessage (int Thread)
    {
    VioWrtTTY("*************************************************\r\n",53,0);
    VioWrtTTY("*                                                *\r\n",53,0);
    VioWrtTTY("*                                                *\r\n",53,0);
    VioWrtTTY("*                                                *\r\n",53,0);
    VioWrtTTY("*                                                *\r\n",53,0);
    if (Thread == 2)
    VioWrtTTY("*              Hello from Thread 2               *\r\n",53,0);
    else
    VioWrtTTY("*              Hello from Thread 3               *\r\n",53,0);
    VioWrtTTY("*                                                *\r\n",53,0);
    VioWrtTTY("*                                                *\r\n",53,0);
    VioWrtTTY("*                                                *\r\n",53,0);
    VioWrtTTY("*                                                *\r\n",53,0);
    VioWrtTTY("*************************************************\r\n",53,0);

    } /* end PrintMessage */
```

Figure 1-13

```
;               Figure 1-13
;               A module definition file for the program of Figure 1-12

NAME FIG1_12 WINDOWCOMPAT

PROTMODE
```

The purpose for using a semaphore in the example program is to assure that only one program thread can call the function **PrintMessage** at a given time. So while one thread is displaying a message box, the other thread does not generate video output.

The example program defines the variable to be used for the semaphore as follows:

```
ULONG Sem = 0;
```

Since this variable is initialized with a zero value, the semaphore is initially in the clear state. Immediately before writing to the screen, both threads call the OS/2 service **DosSemRequest** (explained in Figure 1-14), passing it the address of the semaphore **Sem**. This service performs the following two tasks:

1. If the semaphore is currently set, it suspends the execution of the calling thread until the semaphore is cleared by the other thread.

2. It then sets the semaphore.

After creating a short pause and displaying a message box, both threads call **DosSemClear** (Figure 1-15) to clear the semaphore. Accordingly, while a given thread writes to the screen, the semaphore remains set and the other thread cannot enter the block of code that writes to the screen. The other thread remains blocked in the call to DosSemRequest until the semaphore is cleared. Since the semaphore is initially in the clear state, the first thread to call DosSemRequest is allowed to proceed immediately.

At a given time, only one thread can execute the instructions that lie between the call to DosSemRequest and the call to DosSemClear.

If you run this program, you will see a series of complete message boxes on the screen generated alternately by each of the two threads.

Figure 1-14: The DosSemRequest OS/2 function.

DosSemRequest

☐ *Purpose*: Suspends the current thread, if necessary, until the specified semaphore is clear, and then sets this semaphore.

☐ *Prototype*: UHSORT DosSemRequest

(HSEM hsem,	Semaphore handle; for a RAM semaphore this parameter is the far address of the ULONG variable used for the semaphore; for a system semaphore, it is the handle returned by the **DosCreateSem** or **DosOpenSem** function.
LONG lTimeOut)	The period of time to wait, if necessary, for the semaphore to be cleared; a value of -1 means to wait forever.

☐ *Return Value*: If successful, the function returns zero. If an error occurs, it returns a nonzero error code. The following are the possible error codes:

ERROR_INTERRUPT	Interrupted system call.
ERROR_INVALID_HANDLE	**hsem** parameter contains an invalid handle.
ERROR_SEM_OWNER_DIED	Process that owns semaphore terminated.
ERROR_SEM_TIMEOUT	Time given by **lTimeOut** expired.
ERROR_TOO_MANY_SEM_REQUESTS	Exceeded limit on semaphore requests.

Figure 1-15: The DosSemClear OS/2 function.

DosSemClear

☐ *Purpose*: Clears a semaphore.

☐ *Prototype*: USHORT APIENTRY DosSemClear

(HSEM hsem)	Semaphore handle; for a RAM semaphore this parameter is the far address of the ULONG variable used for the semaphore; for a system semaphore, it is the handle returned by the **DosCreateSem** or **DosOpenSem** function.

☐ *Return Value*: If successful, the function returns zero. If an error occurs, it returns a nonzero error code. The following are the possible error codes:

ERROR_EXCL_SEM_ALREADY_OWNED	Exclusive semaphore already owned by another process.
ERROR_INVALID_HANDLE	**hsem** parameter contains an invalid handle.

Note that the program loops executed by threads 2 and 3 contain two calls to DosSleep. The first call—DosSleep (500L)—simply generates a 1/2 second delay between the appearance of each message box so that you can read the individual messages. After the semaphore is cleared, however, DosSleep is called again, and is passed a value of 0. Passing 0 to this function causes the thread to yield the remainder of its current time slice and allows the system scheduler to run the other thread.

Merely clearing the semaphore does not automatically cause the second thread to begin running; rather, the current thread is allowed to run for the remainder of its time slice. If the semaphore is still clear when the current thread completes its time slice, the system will release the other thread from the call to DosSemRequest and allow it to resume executing instructions. Unfortunately, however, before the current thread completes its time slice, it generally has time to *set* the semaphore by calling

DosSemRequest again at the start of the loop. Consequently, unless the current thread happened to complete its time slice immediately after clearing the semaphore, the other thread would never gain the opportunity to run, and you would see message boxes from only one thread. Calling DosSleep with a zero value allows both threads to run.

The semaphore used in the example program is known as a RAM semaphore. You must explicitly declare the 4-byte (ULONG) variable used for this type of semaphore. OS/2 also provides system semaphores, which are allocated and managed by the operating system—system semaphores are more suitable for coordinating the threads belonging to more than one process.

Semaphores are only one of several forms of interprocess communication provided by OS/2. OS/2 also supports shared memory, pipes, queues, and signals. Note that the general term interprocess communication commonly refers to the exchange of data or the synchronization of activities among individual program threads, as well as among separate processes. For more information on the complex topic of interprocess communication—see a book on basic OS/2 programming or the OS/2 programmer's reference.

CHAPTER 2

PRESENTATION MANAGER PROGRAMS

This chapter continues the discussion of fundamental OS/2 programming techniques by describing the anatomy of a Presentation Manager application—a basic program shell that prints a line of text in a window and displays a simple menu. Although this program performs only the most rudimentary tasks, it meets all the essential requirements for a Presentation Manager application. A program that meets these requirements gains complete access to the vast collection of Presentation Manager functions for managing windows and displaying graphics.

The chapter first summarizes the close relationship between the Presentation Manager and the dynamic-linking mechanism. Next, it presents the set of files used to create the example Presentation Manager application. finally the chapter describes each of these files, first the C source code, and then the supporting files used to build the final program.

This chapter provides only a brief introduction to Presentation Manager programming—a vast topic that could fill many volumes. For further information, see one of the programming guides cited in the Bibliography. The chapter uses a number of terms that have special meanings in the context of the Presentation Manager; these terms are defined in the glossary.

Presentation Manager and Dynamic-Linking

Understanding the Presentation Manager is important for appreciating many of the topics in subsequent chapters. The Presentation Manager and the programs written for it illustrate many of the most important uses for dynamic-link libraries. This section discusses five of these uses.

First, Presentation Manager applications can call most of the basic OS/2 dynamic-link functions described in Chapter 1 (the exceptions will be noted later in the chapter).

Second, the Presentation Manager itself is an operating system extension implemented as a set of dynamic-link libraries. The Presentation Manager is not a freestanding executable program, rather it is a closely integrated set of dynamic-link library functions that can be called by one or more programs (which must meet the requirements that will be discussed). When the first Presentation Manager program is run (normally the user interface shell, which the system automatically runs at boot time), these dynamic-link libraries are loaded into memory and perform initialization tasks. Presentation Manager applications that are subsequently run can share the dynamic-link code that is already loaded. The set of dynamic-link functions that constitute the Presentation Manager is known as the Presentation Manager API. Several of these functions, as well as the general function categories, are described later in the chapter.

The Presentation Manager illustrates that the dynamic-link mechanism can be used to create complex subsystems—complete program environments that smoothly integrate with the operating system. In subsequent chapters, you will discover the features of dynamic-linking that make the creation of such systems possible.

Third, placing sets of the routines you develop within dynamic-link libraries can help to simplify Presentation Manager programming and allow several programs to share the function code. This is a benefit enjoyed by all types of protected-mode programs.

Fourth, in addition to storing the code and data belonging to callable functions, dynamic-link library files can also be used to store OS/2 resources. Resources are a form of read-only data that can be loaded and used for various purposes at program runtime. Although non-Presentation Manager programs can employ certain forms of resources, they are

especially important for Presentation Manager programs. They are used to store strings, menus, dialog boxes, bitmaps, icons, and other objects. To illustrate the methods for defining and using resources, the example program in this chapter includes a Presentation Manager menu.

Finally, dynamic-link files are used to store the character fonts that can be loaded and used to display text within Presentation Manager programs. Although font files are dynamic-link modules, they are given the .FON extension to distinguish them from other types of dynamic-link library files. OS/2 version 1.1 supplies several font files such as COURIER.FON and HELV.FON.

The Source Files

Table 2-1 lists the source files used to create the example Presentation Manager application. As you can see from the names, the files are presented in Figures 2-1 through 2-5.

Table 2-1: Source Files for Presentation Manager

FILENAME	CONTENTS
FIG2_1.C	The C source code.
FIG2_2.RC	The resource script file defining the menu.
FIG2_3.H	Header file containing symbolic constants used for defining and managing the menu; this file is included in FIG2_1.C and FIG2_2.RC.
FIG2_4.DEF	Module definition file for linking the program.
FIG2_5.MAK	Script for preparing the program using the MAKE utility.

These files are typical of the set of source files required to develop a Presentation Manager application; however, some programs may require one or more additional files (for example, files for storing bitmaps or

icons). This chapter explains each of these files: first the C source file and then the four supporting files needed to construct the executable program.

The C Source File

A Presentation Manager application must conform to a specific program architecture. Conformance with this architecture demands that a program perform six basic steps (with some possible variations). These six steps represent the basic shell of a Presentation Manager application; by performing them, a program *becomes* a Presentation Manager application and thereby gains access to all the facilities and services offered by this subsystem. The example program of Figure 2-1 implements each of these operations and therefore provides a simple program skeleton that you can use to develop more complex applications. The following is a summary of these six steps:

1. Call **WinInitialize** to initialize the Presentation Manager for the current process.
2. Call **WinCreateMsgQueue** to create a queue to receive the messages sent to the program window.
3. Call **WinRegisterClass** to register a window class.
4. Call **WinCreateStdWindow** to create a standard window.
5. Process messages sent to the program window.
6. Release Presentation Manager objects before terminating.

The function **main** in Figure 2-1 illustrates these six steps, which will be discussed in order, following Figures 2-1 through 2-5.

Figure 2-1

```
/*
     Figure 2-1

     This is a simple Presentation Manager program, which prints a string in
     the client window and displays a menu;  one menu item displays a message
     box and the other item terminates the program.  Preparing this program
     requires the following files:

          FIG2_2.RC      The resource compiler script defining the menu.
          FIG2_3.H       Header file with symbolic constants for the menu.
          FIG2_4.DEF     Module definition file.
          FIG2_5.MAK     MAKE file for building the program.
*/

#include <OS2.H>
#include <PROCESS.H>

#include "FIG2_3.H"              /* Definitions for menu resource.         */

void Quit (void);                /* Releases PM objects and quits program. */

HWND HFrame;                     /* Frame window handle.                   */
HAB HAncBlk;                     /* Anchor block handle.                   */
HMQ HMesQue;                     /* Message queue handle.                  */

/*** Declare window procedure. ***********************************************/

MRESULT EXPENTRY WndProc (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2);
```

```
void main (void)
     {
     HWND HClient;                  /* Client window handle.                  */
     QMSG QueMess;                  /* Message queue structure.               */
     ULONG WindowSpec =             /* General window specifications.         */
          FCF_MENU           |      /* Create a menu from resource segment.   */
          FCF_MINMAX         |      /* Minimize/maximize box.                 */
          FCF_SHELLPOSITION  |      /* Make window visible on screen.         */
          FCF_SIZEBORDER     |      /* Wide sizing border.                    */
          FCF_SYSMENU        |      /* System menu.                           */
          FCF_TASKLIST       |      /* Display program name in Task Manager.  */
          FCF_TITLEBAR;             /* Title bar.                             */

/*** 1.  Initialize the Presentation Manager and get anchor block handle. ****/

     HAncBlk = WinInitialize        /* Returns an anchor block handle.        */
                    (0);            /* Initialization options: must be 0.     */

/*** 2.  Create a message queue for the current thread. **********************/

     HMesQue = WinCreateMsgQueue /* Returns a message queue handle.           */
                    (HAncBlk,     /* Anchor block handle.                     */
                    0);           /* Minimum queue size: 0 means default size.*/


/*** 3.  Register a window class. ************************************************/

     WinRegisterClass
          (HAncBlk,             /* Anchor block handle.                      */
          "MAIN",               /* Window class name.                        */
```

```
        WndProc,            /* Window procedure associated with class.     */
        0L,                 /* Class style: no styles specified.           */
        0);                 /* Bytes of data storage for each window.      */

/*** 4.  Create a standard window. ***************************************/

    HFrame = WinCreateStdWindow   /* Returns handle to frame window.       */
        (HWND_DESKTOP,          /* Parent window handle.                   */
        WS_VISIBLE,             /* Frame window style.                     */
        &WindowSpec,            /* Address of window specifications.       */
        "MAIN",                 /* Client window class name.               */
        ":  PM Program Shell",  /* Text for title bar.                     */
        0L,                     /* Client window style:  none specified.   */
        0,                      /* Resource module handle: 0 is EXE file.  */
        ID_MENU,                /* Resource identification for menu.       */
        &HClient);              /* Address to receive client window hand.  */

/*** 5.  Process messages. ***********************************************/

    while (WinGetMsg            /* Get messages until WM_QUIT.             */
        (HAncBlk,               /* Anchor block handle.                    */
        &QueMess,               /* Address of message structure.           */
        0,                      /* Window filter: any window.              */
        0,                      /* First message identifier: n/a.          */
        0))                     /* Last message identifier: n/a.           */

        WinDispatchMsg (HAncBlk,&QueMess);      /* Dispatch messages.      */

/*** 6.  Relinquish Presentation Manager objects and terminate application. **/
```

```
     Quit ();
     } /* end main */

/*** The client window procedure and subroutines. ***************************/

MRESULT Paint (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2);

MRESULT Command (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2);

MRESULT EXPENTRY WndProc       /* The client window procedure.              */
     (HWND hwnd,               /* Window handle.                            */
     USHORT msg,               /* The message.                              */
     MPARAM mp1,               /* Message-specific information.             */
     MPARAM mp2)               /* Message-specific information.             */
     {
     switch (msg)
          {
          case WM_PAINT:       /* Process window paint message.             */

               return Paint (hwnd, msg, mp1, mp2);

          case WM_COMMAND:     /* Process message from menu.                */

               return Command (hwnd, msg, mp1, mp2);

          default:  /* Perform the default processing on all other messages. */

               return WinDefWindowProc (hwnd,msg,mp1,mp2);
          }

     } /* end WndProc */
```

```
/*** Function for processing the window paint message. **********************/
MRESULT Paint (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2)
     {
     HPS HPresSpace;          /* Presentation space handle.                  */
     RECTL Rect;              /* Window rectangle, in window coordinates.    */
     static char Message [] = /* Text to display.                            */
          "Hello from the window procedure.";

     /*** Obtain a handle to a presentation space. ***************************/

     HPresSpace = WinBeginPaint    /* Returns presentation space handle.     */
          (hwnd,         /* Window handle.                                   */
          0,             /* 0 requests a cache presentation space.          */
          0);            /* Address of variable to set to invalid region: none.*/

     /*** Obtain the coordinates of the client window. ***********************/

     WinQueryWindowRect
          (hwnd,         /* Window handle.                                   */
          &Rect);        /* Structure to receive coordinates.               */

     /*** Print a line of text. **********************************************/

     WinDrawText  /* Draws a single line of formatted text into a rectangle. */
          (HPresSpace,         /* Presentation space handle.                 */
          0xffff,              /* Length of string: 0xffff means 0 terminated.*/
          Message,             /* Text to be displayed.                      */
          &Rect,               /* Coordinates of rectangle containing text.  */
          CLR_NEUTRAL,         /* Use default foreground color.              */
          CLR_BACKGROUND,      /* Use default background color.              */
                               /* Drawing specifications:                    */
```

```
        DT_ERASERECT |      /* Erase the rectangular area.                */
        DT_LEFT      |      /* Left-justify string within rectangle.      */
        DT_TOP);            /* Place string at top of rectangle.          */

    /*** Release presentation space / revalidate window. ********************/

    WinEndPaint (HPresSpace);          /* Tells Presentation Manager that   */
                                       /* redrawing is complete.            */
    return FALSE;

    } /* end Paint */

/*** Processes menu messages. ***************************************************/

MRESULT Command (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2)
    {
    switch ((int)mp1)
        {
        case ID_HELLO:           /* "Hello" menu item was selected.          */

            WinMessageBox                /* Display a message box.          */
                (HWND_DESKTOP,           /* Handle of parent window.        */
                hwnd,                    /* Handle of owner window.         */
                "Hello from the menu.",  /* Window message.                 */
                "PM Program Shell",      /* Window title.                   */
                0,                       /* ID msg. box window:  n/a.       */
                MB_OK |                  /* Display an "OK" push button.    */
                MB_ICONASTERISK);        /* Display an asterisk icon.       */

            return FALSE;
```

```
        case ID_GOODBYE:
            Quit ();        /* Release PM objects and terminate program.   */
        default:            /* Ignore all other messages.                  */
            return FALSE;
        }
    } /* end Command */

void Quit (void)
/*
    Releases Presentation Manager objects and terminates program.
*/
    {
    WinDestroyWindow (HFrame);          /* Eliminate the window.           */
    WinDestroyMsgQueue (HMesQue);       /* Eliminate the message queue.    */
    WinTerminate (HAncBlk);             /* Sever ties with the PM.         */
    exit (0);                           /* Terminate program.              */

    } /* end Quit */
```

Figure 2-2

```
/*
    Figure 2-2

    Resource script for the program of Figure 2-1
*/
```

```
#include "FIG2_3.H"
 MENU ID_MENU
     BEGIN
     MENUITEM        "~Hello",              ID_HELLO
     MENUITEM        "~Goodbye",            ID_GOODBYE
     END
```

Figure 2-3

```
/*
     Figure 2-3

     Header file to be included in the resource script of Figure 2-2 and the
     C source file of Figure 2-1.
*/

#define ID_MENU     1

#define ID_HELLO    10
#define ID_GOODBYE  11
```

Figure 2-4

```
;               Figure 2-4
;               Module definition file for the program of Figure 2-1
;
NAME                    FIG2_1  WINDOWAPI

PROTMODE
HEAPSIZE        1024
```

Figure 2-5

```
#    Figure 2-5
#    This MAKE file prepares the program of Figure 2-1.  The following files
#    are involved:
#         FIG2_1.C
#         FIG2_2.RC, FIG2_2.RES
#         FIG2_3.H
#         FIG2_4.DEF
#         FIG2_5.MAK
#
FIG2_1.OBJ : FIG2_1.C FIG2_3.H
     cl /c /W2 /Zp /G2ws FIG2_1.C

FIG2_2.RES :  FIG2_2.RC FIG2_3.H
     rc /r FIG2_2.RC

FIG2_1.EXE : FIG2_1.OBJ FIG2_2.RES FIG2_4.DEF
     link /NOI /NOD FIG2_1.OBJ,, NUL.LST, SLIBCE.LIB OS2.LIB, FIG2_4.DEF
     rc FIG2_2.RES FIG2_1.EXE
```

Initialize the Presentation Manager

Before calling any other Presentation Manager functions, a program must call WinInitialize to initialize the Presentation Manager subsystem for the current process, and to obtain an **anchor block handle**. The anchor block handle is a value that identifies the process to the Presentation Manager, and must be passed as a parameter to several other functions.

The Presentation Manager functions are divided into groups of related functions in the same manner as the basic OS/2 services described in

Chapter 1; the function category is likewise identified by a three letter prefix. The two most important groups are the **Win** functions (such as WinInitialize), that manage windows and provide many general-purpose services; and the **Gpi** functions (such as GpiLine), that are used to create graphics output.

The Presentation Manager functions used in the example program are briefly described in Figure 2-6, which lists them in alphabetical order. This figure provides the full function prototypes, but does not attempt to explain the use of each of the parameters and return values. Also, the chapter text explains only those parameters that relate to the topic under discussion. For a complete description of the functions, see the OS/2 Presentation Manager programmer's reference. See also the comments in the source listing of Figure 2-1 for a brief explanation of how each parameter is used in the example program.

Figure 2-6: PM functions used in Figure 2-1's example program.

WinBeginPaint

☐ Supplies a handle to a presentation space associated with the specified window, so that the window procedure can update the window display:

```
HPS APIENTRY
WinBeginPaint
(HWND hwnd,
HPS hps,
PRECTL prclPaint)
```

WinCreateMsgQueue

☐ Creates a queue to receive the messages sent to the program window:

```
HMQ APIENTRY WinCreateMsgQueue
(HAB hab,
SHORT cmsg)
```

WinCreateStdWindow

☐ Creates a "standard window," which consists of a set of related Presentation Manager windows:

```
HWND APIENTRY WinCreateStdWindow
(HWND hwndParent,
ULONG flStyle,
PULONG pflCreateFlags,
PSZ pszClientClass,
PSZ pszTitle,
ULONG styleClient,
HMODULE hmod,
USHORT idResources,
PHWND phwndClient)
```

WinDefWindowProc

☐ Performs default message processing; the window procedure calls this function to handle messages it does not want to process itself:

```
MRESULT APIENTRY WinDefWindowProc
(HWND hwnd,
USHORT msg,
MPARAM mp1,
MPARAM mp2)
```

WinDestroyMsgQueue

☐ Eliminates a message queue created by the function WinCreateMsgQueue:

```
BOOL APIENTRY WinDestroyMsgQueue
(HMQ hmq)
```

WinDestroyWindow

☐ Eliminates a Presentation Manager window:

```
BOOL APIENTRY WinDestroyWindow
(HWND hwnd)
```

WinDispatchMsg

☐ Passes a message to the system for processing; the system invokes the window procedure belonging to the target window:

```
MRESULT APIENTRY WinDispatchMsg
(HAB hab,
PQMSG pqmsg)
```

WinDrawText

☐ Displays text string within a window, formatting the string within specified rectangle:

```
SHORT APIENTRY WinDrawText
(HPS hps,
SHORT cchText,
PCH lpchText,
PRECTL prcl,
LONG clrFore,
LONG clrBack,
USHORT rgfCmd)
```

WinEndPaint

☐ Releases presentation space obtained from WinBeginPaint; the window procedure should call this function when it has updated the window display:

```
BOOL APIENTRY WinEndPaint
(HPS hps)
```

WinGetMsg

☐ Extracts a message from the program message queue:

```
BOOL APIENTRY WinGetMsg
(HAB hab,
PQMSG pqmsg,
HWND hwndFilter,
USHORT msgFilterFirst,
USHORT msgFilterLast)
```

WinInitialize

☐ Initializes the Presentation Manager for the current process:

```
HAB APIENTRY WinInitialize
(USHORT fOptions)
```

WinMessageBox

☐ Displays a temporary window containing the specified message, and pauses for user input:

```
USHORT APIENTRY WinMessageBox
(HWND hwndParent,
HWND hwndOwner,
PSZ pszText,
PSZ pszCaption,
USHORT idWindow,
USHORT flStyle)
```

WinQueryWindowRect

☐ Obtains the current dimensions of a window:

```
BOOL APIENTRY WinQueryWindowRect
```

```
(HWND hwnd,
PRECTL prclDest)
```

WinRegisterClass

☐ Registers a window class and assigns a window procedure to this class. This procedure will process the messages sent to any window created as a member of the class:

```
BOOL APIENTRY WinRegisterClass
(HAB hab,
PSZ pszClassName,
PFNWP pfnWndProc,
ULONG flStyle,
USHORT cbWindowData)
```

WinTerminate

☐ Relinquishes access to the Presentation Manager; called immediately before program termination:

```
BOOL APIENTRY WinTerminate
(HAB hab)
```

Note that unlike the basic OS/2 functions described in Chapter 1, the Presentation Manager functions do not return specific error codes to the calling program. Many of these functions use the return value to communicate information other than the error status. If an error occurs, a Presentation Manager function will return a special value such as FALSE, 0, or NULL, indicating that some error occurred (the special values are documented for each function in the programmer's reference). To obtain a code indicating the specific error, you can then call the function **WinGetLastError**.

Create a Message Queue

All Presentation Manager programs that display a window must call the WinCreateMsgQueue function to create a message queue. The system places messages in this queue to inform the program of a wide variety of important events; for example, it sends a message when the window data requires updating and when a character is received from the keyboard. Each message placed in the queue is addressed to a specific window. You will see shortly how to extract these messages from the queue and how to process them. Note that WinCreateMsgQueue is passed the anchor block handle received from WinInitialize.

Register a Window Class

To be able to process messages, a Presentation Manager program must call WinRegisterClass to register a window class. You must pass this function the name of the class you are creating. You can choose any name for this class; the example program specifies the name "MAIN." You must also pass the address of the program function that will process messages (specifically, those messages sent to any window belonging to the class you are registering; in the example program, there is only one such window—**client window**). The example program passes the address of **WndProc**, which is the message processing function defined within the C source file. Since messages are addressed to specific windows, this function is known as a **window procedure**; in a C program, it is an appropriately declared C function.

The essential task performed by WinRegisterClass is to associate the address of a window procedure with a class name. In the next section, you will see how this class name is used.

Create a Standard Window

Once you have established a message queue and have registered a window class, you are ready to call WinCreateStdWindow to create a standard program window. The "standard window" created by this function is actually a collection of related windows; each visible component, such as the title bar and window border, is a distinct Presentation Manager window.

WinCreateStdWindow automatically creates a **frame window**, which underlies the entire area occupied by the standard window collection, but has no visible characteristics itself—this window is known as the **parent** of the other windows. You must, however, specify each of the other component windows you want included. The example program selects the five windows shown in Table 2-2 by setting the appropriate bits of the variable pointed to by the third parameter. In the example program, the variable is named **WindowSpec** and is initialized with the appropriate bitmasks at the beginning of main:

Table 2-2: Presentation Manager Windows

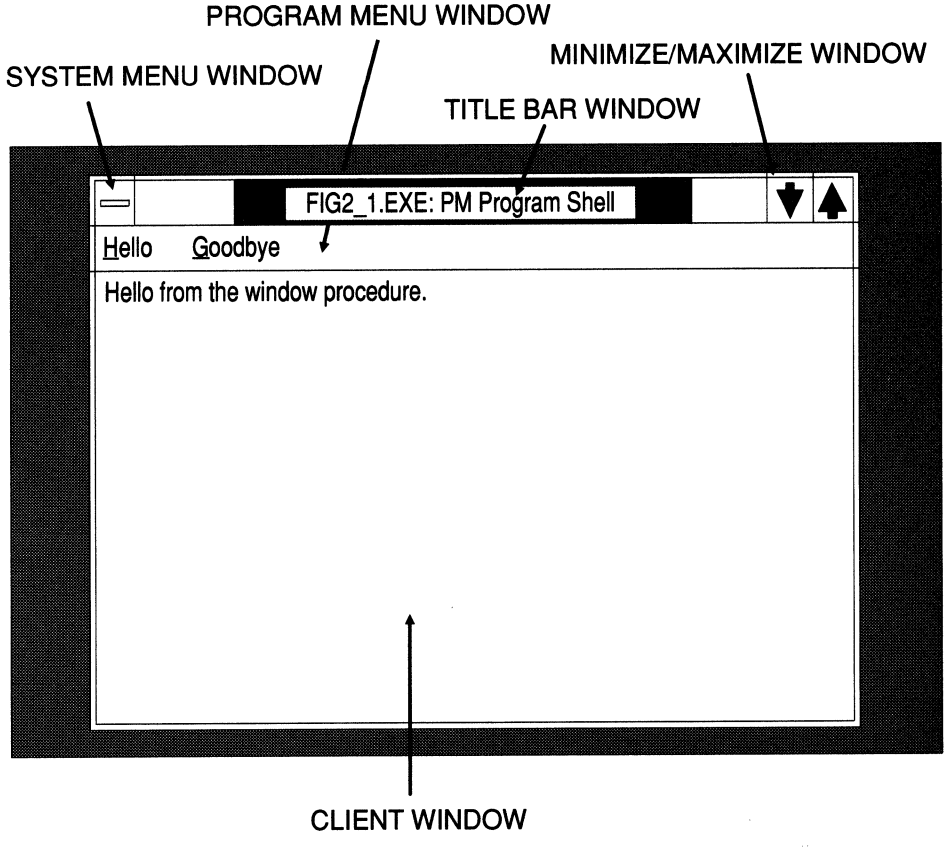
BITMASK	COMPONENT WINDOW CREATED
FCF_MENU	A custom menu (the definition for this menu is stored as a program resource).
FCF_MINMAX	A minimize/maximize box, used to minimize the window to an icon or to maximize it to encompass the entire screen.
FCF_SIZEBORDER	A sizing border, which is a large border that can be used to change the size of the window with a mouse.
FCF_SYSMENU	A system menu, which gives the user access to commands for moving the window, closing the program, or performing other system operations.
FCF_TITLEBAR	A title bar, used to display the program name and to move the window with a mouse.

Note that setting the FCF_MENU bit causes the Presentation Manager to load a menu from a resource that is stored within the program, and to display this menu near the top of the window. If you request the Presentation Manager to load a menu, you must supply the identifier of the appropriate resource as the eighth parameter (**idResources**); this identifier is initially assigned when you define the menu. The section on The

Resource Script, later in the chapter, describes how to define a menu and how to store the menu definition as a program resource.

Figure 2-7 illustrates the standard window created by the example program and labels each of the component windows. Note that there are several other bitmasks assigned to WindowSpec, which do not specify component windows rather, they control other aspects of the standard window. See the programmer's reference for details.

Figure 2-7: The standard window created by the example program.



You can see that the call to WinCreateStdWindow has created six windows: the frame window and the five component windows listed in Table 2-2. This function call creates one more important window: the client window. The client window occupies the open area inside the other component windows and is the window in which a program normally displays its data. The fourth parameter (**pszClientClass**) specifies the name of the class for the client window (if you pass a NULL value, *no* client window is created). The example program passes the string "MAIN," which is the name of the class that was registered in the call to WinRegisterClass. As a result of assigning the client window to the class "MAIN," all messages sent to the client window will be processed by the function that was associated with this class when it was registered: WndProc. As you will see in the next section, WndProc serves to display data within the client window and to respond appropriately when a menu item is selected.

Process Window Messages

As soon as the main function has created the standard window, it must immediately begin dispatching the messages addressed to the component windows. The system sends a barrage of messages to these windows to inform them of relevant events. For example, the Presentation Manager sends a message to the system menu window when the user clicks the mouse within its borders, and it sends a message to the client window when the user selects a menu item or the data in this window requires updating.

The system places these messages in the message queue created by the call to WinCreateMsgQueue. To handle the messages, the main function enters a program loop that repeatedly extracts a message from the queue and then dispatches this message for processing. A message is extracted from the queue by calling the **WinGetMsg** function; this message is received in a **QMSG** structure (**QueMess**). In general, WinGetMsg extracts messages from the queue in the same order in which they were inserted— there are some exceptions. The program loop then immediately dispatches the message by calling **WinDispatchMsg**.

When the program passes a message to WinDispatchMsg, the system calls the window procedure belonging to the target window. Remember

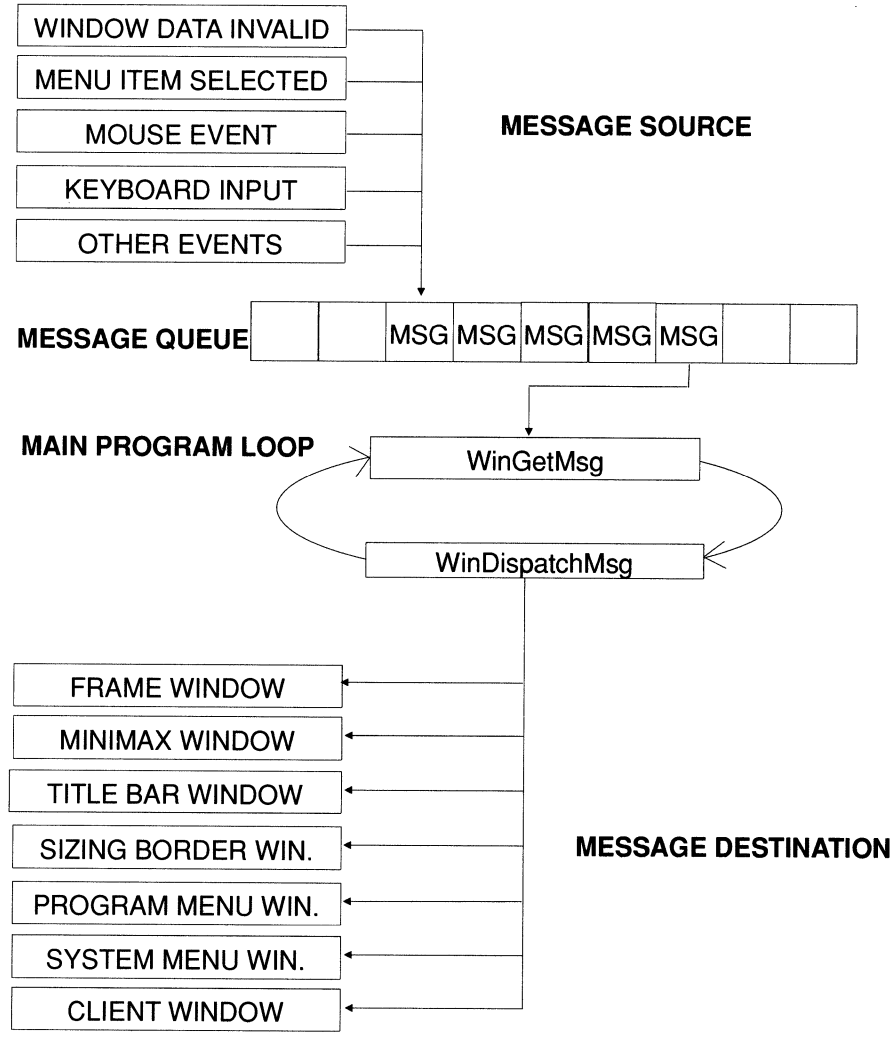
that a message is addressed to a specific window; one of the fields of the QMSG structure holding the message contains the identifier of the target window. The system passes the contents of the message as parameters to the window procedure, which performs the actual message processing. The parameters passed to this procedure are described later.

The window procedure associated with a given window depends upon the window's class (every window belongs to a class, which specifies the window procedure and possibly one or more other attributes). All of the component windows created by the call to WinCreateStdWindow—except the client window—belong to default window classes defined by the system. Accordingly, the messages that are sent to each of these windows are processed by default window procedures *supplied by the system*. Therefore, the behavior of these windows is not under the program's control.

Remember that in the example program the client window was assigned to the class "MAIN"; when this class was created in the call to WinRegisterClass, it was associated with the program function WndProc. When WinDispatchMsg receives a message addressed to the client window, it immediately calls WndProc, passing it the contents of the message. You will see shortly how WndProc processes messages it receives.

Figure 2-8 illustrates how messages are processed in a Presentation Manager program. Note that the message processing loop in the function main continues running for the duration of the program. The loop terminates when WinGetMsg extracts the WM_QUIT message, which the system places in the queue when the user selects the Close item from the system menu; this message causes WinGetMsg to return FALSE. Message processing is the core of a Presentation Manager application. Once the program has performed necessary initializations and has created a window, it assumes a passive role. Most subsequent program activity takes place within the window procedure in response to window messages.

Figure 2-8: Message processing in a Presentation Manager program.



The Window Procedure In the example program the window procedure WndProc is declared as follows:

```
MRESULT EXPENTRY WndProc
(HWND hwnd,
USHORT msg,
MPARAM mp1,
MPARAM mp2);
```

WndProc is called whenever a message is sent to the client window. The four parameters passed to this function contain the essential contents of the message; they hold the values that are listed in Table 2-3:

Table 2-3: Parameter Values

PARAMETER	VALUE
hwnd	Handle of the window to which the message is addressed.
msg	The identifier of the message; the header files contain symbolic constants for these identifiers, such as WM_PAINT, the identifier of the message sent when the window needs redrawing.
mp1	Message-specific information.
mp2	More message-specific information.

The first task of the window procedure is to branch to the appropriate routine to process the specific message—indicated by the message identifier in *msg*. WndProc processes two messages: WM_PAINT, which is sent whenever the data displayed in the window needs to be redrawn, and WM_COMMAND, which is sent whenever the user selects an item from the program menu—*not* the system menu. The system sends a myriad of other messages to the client window; rather than processing these messages. However, WndProc passes them to the **WinDefWindowProc** function, which performs minimal default processing.

In response to the WM_PAINT message, the window procedure calls the function **Paint**, passing it all four message parameters. Paint is the function that creates the program display within the client window. In this simple example, it erases the current contents of the window and prints a string. In a more complex application, such as a word processor, it could display an elaborate combination of text and graphic images.

It might seem unusual that window data is drawn in response to a message from the system instead of when the program needs to create or update its display. However, the system sends this message at propitious times, namely, when the window is first created, and subsequently, whenever the window data needs regenerating due to the actions of the user or of another program. For example, when a portion of the window is uncovered, or when the size of the window is increased. Also, a program can explicitly cause the system to send the WM_PAINT message whenever it needs to update its display—by calling the **WinInvalidateRect** and **WinUpdateWindow** functions. See the programmer's reference.

Writing data to the client window requires several steps, because under the Presentation Manager you do not draw directly onto a physical device, such as the screen or a printer, but rather onto an abstract surface known as a **presentation space**. A presentation space is implemented as a data structure within the Presentation Manager, and has a set of characteristics that are independent of any physical device, such as a current font and drawing colors. To produce actual output, the presentation space must be associated with a physical device. To obtain a presentation space and to display window data through this presentation space, WndProc performs the following steps:

1. It calls **WinBeginPaint** to obtain the handle to a presentation space that is already associated with the client window.

2. It calls **WinQueryWindowRect** to obtain the current dimensions of the client window.

3. It calls **WinDrawText** to print a string within the window. The string is displayed within the rectangle specified by the fourth parameter; this parameter is passed the dimensions of the entire client just obtained from WinQueryWindowRect. The last parameter is assigned the flags DT_LEFT and DT_TOP, which place the string at the upper-

left corner of the rectangle, and the flag DT_ERASERECT, which causes the function to erase the rectangular area before printing the string.

4. It releases the presentation space by calling **WinEndPaint**.

Note that due to the special input/output requirements of Presentation Manager programs, they should *not* call the standard Vio, Kbd, and Mou functions described in Chapter 1, which are designed for non-Presentation Manager programs. Exceptions are Presentation Manager programs that create an *advanced* Vio presentation space, which allows them to use most of the Vio functions. See the programmer's reference for more information. Presentation Manager programs, however, can call almost all of the standard Dos API functions.

In response to the WM_COMMAND message, the window procedure calls the function **Command**. When the user selects a menu item, the system sends this message to the client window, passing the identifier of the selected item in the *mp1* parameter. The identifiers for the menu items are initially chosen when the menu is defined (defining a menu is discussed in the section on The Resource Script, later in the chapter). The menu displayed by the example program has two items—labeled "Hello" and "Goodbye." If the user has selected the "Hello" item, *mp1* will contain the identifier of this item, ID_HELLO, and Command calls **WinMessageBox** to display a **message box**. A message box is a temporary window that displays a string and pauses for user input. If the user has selected the "Goodbye" item, *mp1* will contain the identifier ID_GOODBYE, and the function calls **Quit** to terminate the program (Quit is described in the next section).

Release Presentation Manager Objects

Finally, before a Presentation Manager terminates, it should release the various objects it has allocated from the system. The example program calls the function Quit, defined at the end of the source file of Figure 2-1, which releases these objects. (Quit is called either when the user selects the "Goodbye" program menu item, or when the user selects the Close item from the system menu. As mentioned previously, closing the program

through the system menu results in the termination of the program loop in main and causes control to pass to the call to the function Quit.

Quit calls **WinDestroyWindow** to eliminate the program window, it calls **WinDestroyMsgQueue** to eradicate the message queue, and it calls **WinTerminate** to sever its final ties with the Presentation Manager. Releasing these objects is not mandatory, since the system releases them automatically when the program terminates. However, explicitly releasing them in the order shown can give the program termination a more orderly appearance as various objects are removed from the screen.

The Supporting Files

This section describes the files that are used in conjunction with the C source file to generate the executable Presentation Manager program. The actual steps for preparing the program from these files are outlined in the description of the MAKE file.

The Resource Script

Figure 2-2 listed the resource script that is used to define the program menu. As you will see in the description of the MAKE file, the resource script is processed by the Microsoft resource compiler. This utility converts the resource script into binary data and inserts this data into a resource segment within the executable program file. When you call WinCreateStdWindow to create a standard window and specify the FCF_MENU option, the system reads the menu definition from the resource segment and displays the menu near the top of the window.

A menu definition must begin with the keyword MENU, followed by the menu identifier. You can select any reasonable value for the identifier, which is used to specify the menu resource in the call to WinCreateStdWindow (as the eighth parameter, idResources). The example application uses the value ID_MENU, which is defined in the header file.

The definitions of the menu items are surrounded by the keywords BEGIN and END. The definition of each item begins with the keyword MENUITEM, which is followed by a string containing the actual text to appear on the menu. The tilde character (~) in the text string is not

printed literally; rather, it causes the system to underline the following character, and allows the user to select the menu item by typing the underlined character. The definition concludes with the item identifier. The two items defined for the example program are given the identifiers ID_HELLO and ID_GOODBYE. As described in the previous section, the item identifier is sent with the WM_COMMAND message to inform the client window that the user has selected the corresponding item. The menu displayed by the example program is illustrated in Figure 2-7.

The Header File

The header file in Figure 2-3 defines the symbolic constants used to identify the menu resource and the individual menu items; it is included in the C file of Figure 2-1 and in the resource script of Figure 2-2. Since these constants are used in both the resource script and in the C source file, it is convenient to place them in a single header file included in both source files. Using a separate header is especially important in a complex Presentation Manager application that uses many resource definitions.

The Module Definition File

The use of a module definition file for linking an OS/2 program was introduced in Chapter 1. Figure 2-4 provides a module definition file for the example Presentation Manager program; this file contains four new items not part of the definition file given in Chapter 1. First, program type specified by the NAME statement is WINDOWAPI, which indicates that the program is a full Presentation Manager application, rather than WINDOWCOMPAT, which indicates that the program can run within a window managed by the system. Second, the HEAPSIZE command specifies a program heap size of 1,024 bytes. The program heap is an area at the end of the default data segment used for dynamic-memory allocation. Third, the STACKSIZE command specifies a 4,096 byte-program stack; the stack is also located within the default data segment—immediately below the area reserved for the heap.

Finally, and most importantly, the definition file contains an EXPORTS statement. When linking a Presentation Manager application, *you must list the names of all window procedures in an EXPORTS statement*. The

names must be given in all uppercase letters, because window procedures conform to the Pascal naming convention, according to which the compiler converts the function names to uppercase letters when writing them to the object file. Specifying a function in an EXPORTS statement renders that function accessible to the functions of the Presentation Manager dynamic-link modules. As you will see in Chapter 4, this statement is also used to list the dynamic-link functions when linking a dynamic-link library.

The MAKE File

Finally, Figure 2-5 provides a MAKE file for preparing the executable program. When compiling the C source file, the MAKE file specifies the following two command line switches in addition to those used to prepare the protected-mode programs in Chapter 1:

- **/Gw** This option should be used for preparing a Presentation Manager program that contains a window procedure. It causes all functions to save the value of the DS register, set DS to the segment selector for the C data segment, and to restore the original DS value before returning. These steps allow the window procedure to access the C data segment when called by a function within the Presentation Manager dynamic-link modules, that use a *different* data segment.

- **/Gs** This option disables the calls made to the C stack-checking routine at the beginning of each function; the C stack-checking routine is not compatible with the Presentation Manager since it attempts to write to the screen using conventional teletype-style screen output, which is not allowed within the Presentation Manager windowed environment.

Notice that the resource segment containing the menu definition is prepared with the resource compiler (RC.EXE) in two distinct steps:

1. The resource compiler is invoked with the /r flag to convert the resource script, FIG2_2.RC, to a file containing the resource data in binary format, FIG2_2.RES. The /r flag means to compile the resource script without performing step 2.

2. After the new executable file FIG2_1.EXE has been generated by the linker, the resource compiler is called again (without the /r flag) to insert the resource data from the file FIG2_2.RES into a resource segment within the EXE file.

You could also perform steps 1 and 2 by issuing the following single command *after* the program is linked:

```
rc FIG2_2.RC FIG2_1.EXE
```

By performing these two steps separately, however, you can avoid recompiling the resource script (which is the lengthier of the two operations) if it has not been changed since it was last compiled.

The other commands and switches in this MAKE file are the same as those described in Chapter 1.

CHAPTER 3

HOW DYNAMIC-LINK LIBRARIES WORK

Now that you have gained an overview of basic OS/2 programming techniques and you have seen how to use dynamic-link libraries within an application program, this chapter describes how dynamic-link libraries work.

The first section gives a detailed description of the dynamic-linking mechanism, discussing each process that occurs when you compile, link, and run a program that calls one or more dynamic-link library functions. The second section summarizes the basic uses for dynamic-linking and its advantages over static (that is, conventional) linking.

This chapter is unique; unlike the other chapters, it describes mechanisms and processes rather than presenting techniques. It provides basic theoretical information important for understanding the methods for creating the dynamic-link libraries given in the remainder of the book.

The Process

This section describes the processes that occur during each phase of preparing and loading an application program that contains one or more

calls to dynamic-link library functions. The discussion of the dynamic-linking mechanism is divided into the following stages:

- Compiling the program
- Linking the program
- Loading the program
- Calling the dynamic-link function
- Terminating the program

This chapter describes the most common method of dynamic-linking—known as **loadtime dynamic-linking**—in which dynamic-link libraries are automatically loaded when the program is run. A program can also explicitly load selected dynamic-link modules, and obtain the addresses of the required functions at any point while the program is running. This alternative process is known as **runtime dynamic linking** and is discussed in Chapter 8.

Compiling the Program

Chapters 1 and 2 described how to call the dynamic-link functions provided by the operating system. As you saw in those chapters, a dynamic-link function is called in the same manner as other external functions. (Remember that an external function is one that is called by the program but is not contained within the current source code file. It can be, for example, a C library function, a function you have written within another C or assembly language source file, or a function in a dynamic-link library.) When the compiler encounters a call to an external function, it cannot supply the actual address of the function for the CALL instruction; rather, it leaves the address unspecified and places an external reference for the function within the object file. (Note that if a given source file has several calls to the same external function, the compiler writes only a single external reference.) The external reference consists of the name of the function and an optional type description, and is located within a record that lists all external symbols for the current object

module. The object module is either a freestanding object file or an object file that has been added to a library file.

Since the compiler processes only a single source file at a time, it cannot check whether the external function actually exists; it merely places an external reference in the object file, assuming that the linker will be able to resolve this reference when the object file is linked. To resolve an external reference means to determine the location of the actual function code.

The compiler processes a call to a dynamically linked function in exactly the same way that it processes a call to a conventionally linked function. Both calls simply generate an external function reference in the resulting object file. The compiler does not know about dynamic-link functions.

Linking the Program

The differences between static and dynamic linking first appear when the linker processes an external function reference in the program object file. The external reference itself does not indicate the method of linking; rather, the method of linking is determined by the way that the linker resolves the reference. When the linker attempts to resolve an external reference, it begins searching the following files:

- Any other object modules (.OBJ files) specified on the LINK command line.
- Any library files (.LIB files) specified on the LINK command line (standard library files or import libraries).
- Any import definitions given in the module definition file (.DEF file), if a definition file has been specified on the LINK command line. (Import definitions are given in an IMPORTS statement and resolve references only to dynamic-link functions; this statement is discussed later in the chapter).

For example, given the LINK command line:

```
link /NOI /NOD APP.OBJ MOD1.OBJ,,,SLIBCE.LIB OS2.LIB, APP.DEF
```

the linker would search the object file MOD1.OBJ, the standard library file SLIBCE.LIB, the import library file OS2.LIB, and the IMPORTS statement (if any) in the module definition file APP.DEF.

As the linker searches these files, it may first find a conventional object module that resolves the reference, or it may first find a special dynamic-link object record or IMPORTS definition that resolves the reference. If the linker is unable to find any of these items, it prints an "Unresolved external" error message (also, under OS/2, it marks the program file as non-executable).

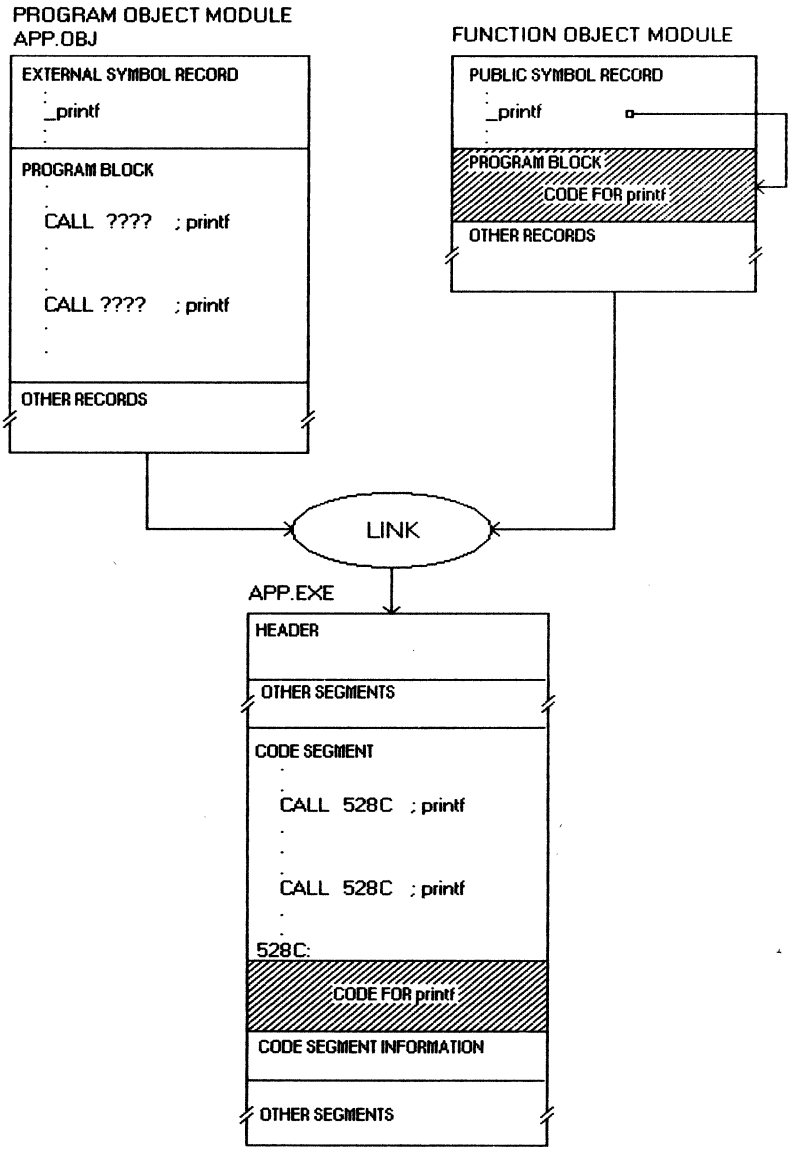
If the linker first finds a conventional object module that resolves the external reference, then the function is linked statically. Such an object module may be a separate .OBJ file or it may be part of a standard library file. A given object module contains a record of all public symbols defined in the module; if the name of the external function is found in this record, then the module is used to resolve the reference. To resolve a function reference using a conventional object module, the linker performs the following two steps:

1. It copies the entire body of code and data defined in the module directly into the executable program file (provided, of course that the code and data have not already been copied into the file to resolve a previous external reference). In addition to the desired function, the object module may contain one or more other functions and variables; all of these items are written to the executable file since the object module is the smallest linkable unit.

2. It writes the actual function address to the address portion of the CALL instruction that invokes this function. More precisely, it supplies the offset address; if the function is invoked through a far CALL, the segment portion of the address cannot be furnished until the program is run.

Figure 3-1 illustrates how the linker resolves references to a statically linked function.

Figure 3-1: Resolving references to a statically linked function.



Alternatively, the function will be dynamically linked if the linker first finds one of the following two items as it attempts to resolve the external function reference:

- A dynamic-link record for the external function in an import library.
- An IMPORTS statement in the module definition file that includes an import definition for the external function.

Chapter 1 described how to use an import library to resolve references to the OS/2 API dynamic-link functions. Rather than containing object modules that supply the complete body of code and data for each function, import libraries contain simple dynamic-link records for each function. A dynamic-link record is a special object record (unique to OS/2) that contains the following information:

1. The function name, as it appears in the program source file and in the external reference in the program object file. This is known as the **external name.**
2. The name of the dynamic-link library file that contains the actual function code and data.
3. The entry point of the function within the dynamic-link library. The entry point may be given as a simple number (the ordinal value of the function), or as an entry point name. The entry point name may be the same as the external name (item 1), or it may be a different name.

You will see shortly what the linker does with this information. Chapter 4 describes how to prepare an import library for the dynamic-link libraries you create, using the IMPLIB utility. **Import library** refers to a library file containing import definitions created with the IMPLIB utility. However, you can also add standard object modules to such a library using the LIB utility. A given library file can contain both standard object modules, which are recognized by the presence of certain object records, and dynamic-link object records. The linker simply searches all specified libraries and responds according to the types of the individual records it finds in these files.

The linker can also resolve an external function reference for a dynamic-link function through an import definition in a module definition file. Import definitions are contained in an IMPORTS statement, which is described in Chapter 4. As you will see in Chapter 4, import definitions supply the same three items of information given by dynamic-link object records (listed above), and provide an alternative to using an import library.

When the linker resolves an external reference to a dynamic-link function either through a dynamic-link object record or through an import definition, it writes the following three items of information to a relocation record within the program file. The relocation records for a given segment are found in a relocation table that follows the segment image in the executable file:

1. The offset of the reference to the dynamic-link function within the program code segment; in other words, the offset of the address portion of the far CALL instruction that invokes the function. (Note that only the offset of the reference need be stored since each program segment has its own relocation table, and thus the segment is known implicitly.)

2. The name of the dynamic-link library file containing the function.

3. The entry point of the function within the dynamic-link library, specified either as an ordinal value or as a name—but not both.

The three values written to the relocation record correspond to the three items specified by the dynamic-link object record or the import definition. The names are not written directly to the relocation record; rather, all name strings are stored in a single imported name table, and the appropriate index to this table is written to the relocation record. This structure saves space, since module or function names that appear more than once do not need to be duplicated.

Thus, unlike the static linking mechanism, the function code and data are not read into the program file, and the address portion of the CALL instruction is not supplied. Instead, a relocation record is established in the program file, which contains the information required to locate the dynamic-link function at load-time and a pointer to the address field of the CALL instruction within the program code.

Note that the linker creates a new relocation record only for the first call to a given dynamic-link function that occurs within a code segment. If a second call to this same function is encountered, the linker places the offset of the address portion of the second call within the address portion of the first call. Thus, it forms a linked-list of references to the dynamic-link function, and continues to add subsequent calls to the same function to the end of the list. For example, if a code segment contains calls to a given dynamic-link function at offsets 0x143c, 0x2482, and 0x5f9a, the relocation record would contain the offset of the address field of the first CALL instruction—0x143d. Note that the address begins one byte beyond the beginning of the instruction. The three CALL instructions would be assigned the values that are recorded in Table 3-1:

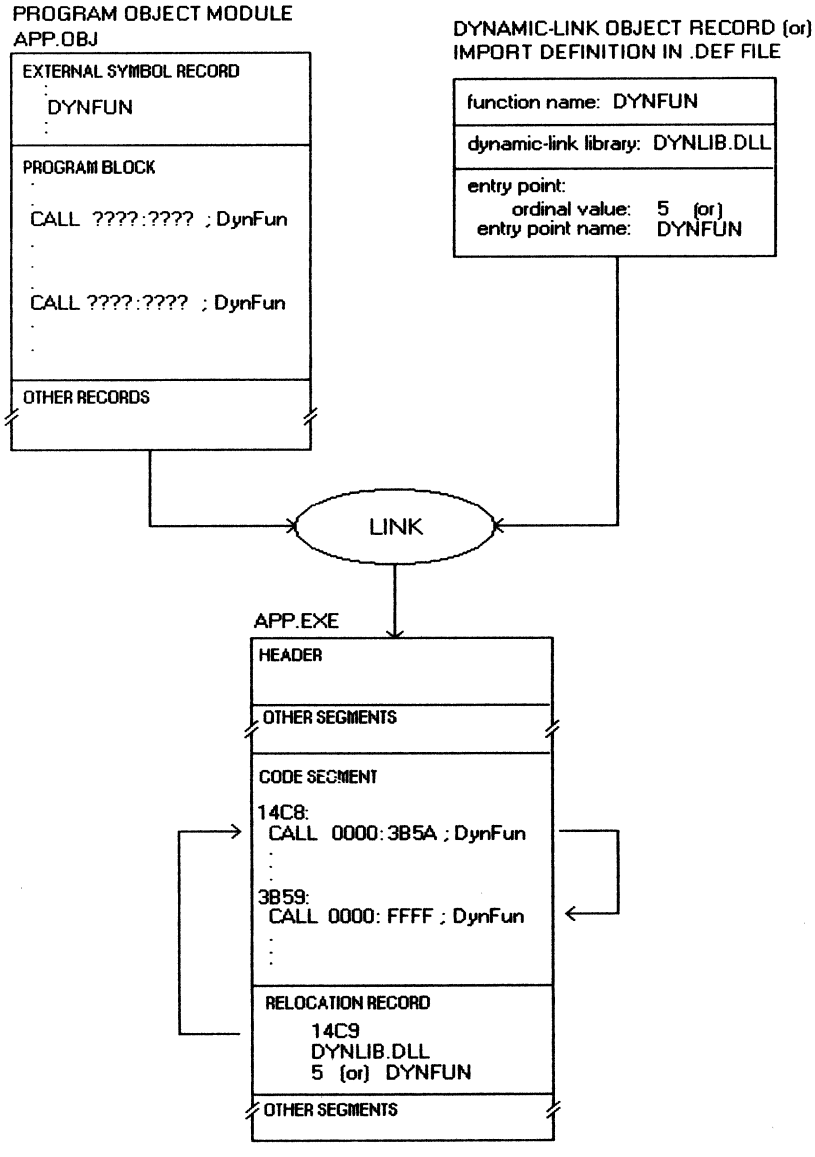
Table 3-1: CALL Instructions

OFFSET OF CALL	INSTRUCTION
0x143c	CALL 0000:2483
0x2482	CALL 0000:5f9b
0x5f9a	CALL 0000:ffff

The value, oxfffff, is a special value indicating the end of the list. Obviously, the program is not intended to run with these address values. As you will see in the next section, the linked list they establish will be used by the program loader to fill in the appropriate address at all required locations in the code.

Figure 3-2 illustrates the process of resolving the references to a dynamically linked function, and shows the linked list that connects the CALL instructions. (Compare this illustration to Figure 3-1, which depicts the process of resolving statically linked functions. Note that since a given program can contain calls to both statically linked and dynamically linked functions, the processes illustrated in both of these figures can take place during the linking of a single program.)

Figure 3-2: Resolving references to a dynamically linked function.



Loading the Program

Because statically linked functions form an integral part of the code and data contained in an executable file, they are automatically loaded when the program is run. Also, since the offset addresses in all CALL instructions to statically linked functions were supplied by the linker, the loader does not have to adjust these addresses. (Note, however, that if a function is invoked through a far CALL instruction, the loader will have to supply the segment portion of this address, since the segment values are unknown until the program is loaded.) Figure 3-3 illustrates the loading of a statically linked function; this figure is presented for comparison with Figure 3-4, which shows the process of loading a dynamically linked function.

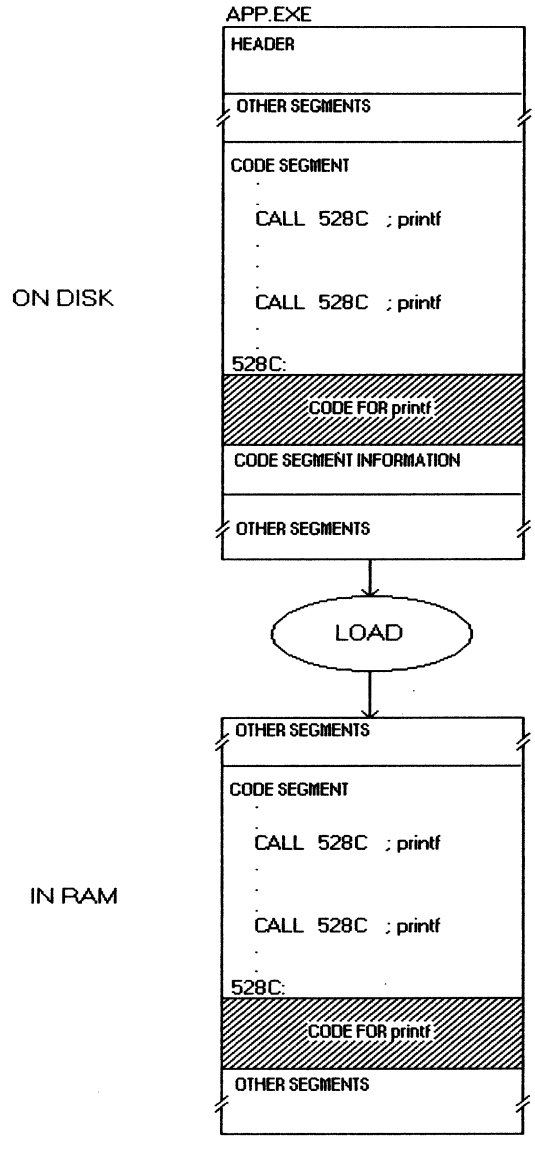
In contrast, if the program contains dynamically linked functions, the loader must perform several important tasks to process these functions. As explained in the previous section, the executable file contains a relocation record for each dynamic-link function called by the program. Accordingly, when the program is run, the loader goes through the relocation table for each segment, and performs the following two basic operations for each relocation record that refers to a dynamic-link function:

1. It loads the dynamic-link library code and data into memory.
2. It writes the full address of the dynamic-link function in memory to all CALL instructions that invoke this function.

These two tasks are now discussed individually.

Loading Dynamic-Link Libraries The loader encounters a relocation record for a dynamic-link library that has not already been loaded into memory. It loads the entire body of code and data contained in this library (this is analogous to the static linking process in which the linker always links the entire object module into the program). As mentioned previously, the name of the library is contained in the relocation record; the loader searches for this library in all directories specified by the LIBPATH configuration command.

Figure 3-3: Loading a statically linked function.



In addition to loading the segments of the dynamic-link library into memory, the linker must render these segments accessible to the calling process. As you will see in the discussion of virtual memory in Chapter 5, to allow a process to access a given segment, the system must establish a segment descriptor which belongs to the process and defines the properties of the segment, such as its physical address and whether it is a code segment or a data segment.

What does the loader do if the dynamic-link library has already been loaded into memory during the loading of a previous process that referenced this library? Its action depends upon the specific dynamic-link library segment. If the segment is marked as a code segment, the loader does not load a new copy into memory; rather, the new process shares the code that has already been loaded. Any number of processes can share a single code segment because such segments cannot be written to by any process; thus, one process cannot alter or accidently corrupt the code used by another process.

If the segment is marked as a data segment, the loader may or may not load a new copy from the dynamic-link library file. As you will see in Chapter 5, you can specify in the module definition file whether a new copy of a given data segment is loaded for each process (an instance data segment), or whether all processes using the dynamic-link library share the original copy of the segment in memory (a global data segment). In general, instance data segments are more common and are easier to use than global segments. Instance data segments prevent data conflicts among separate processes using the dynamic-link library (the client processes), and normally permit you to write a dynamic-link function like a normal subroutine that is called by a single process. Global data segments allow the dynamic-link function to share data among multiple client processes, and force the function to keep track of its individual clients. The use of global segments is discussed in Chapter 5.

Note that a dynamic-link library may contain references to other dynamic-link libraries. As described in Chapter 4, a dynamic-link library is prepared by the linker and contains relocation records conforming to the same format as a normal executable file. If the loader, while loading a dynamic-link library, discovers a relocation record within this library that refers to another dynamic-link file, it immediately begins loading the

newly referenced file, and when it has completed loading this file, it resumes processing the original dynamic-link library. Thus, the loading process can be recursive, and there is no documented limit to the level of recursion. As illustrated later in the chapter, references among dynamic-link libraries can even be circular.

Once the loader has copied a dynamic-link library into memory, it may execute an initialization routine contained within this library. Some dynamic-link libraries do not contain initialization routines, some contain initialization routines that are executed only when the first client process is run, and some contain initialization routines that receive control each time a new client is run. Chapter 6 discusses the techniques for writing both types of initialization routines. This routine runs before the loader has completed loading the client process. Initialization routines are especially important for dynamic-link subsystems that need to initialize a shared device or other object.

Finally, when a program or dynamic-link library segment is "loaded" into memory, the appropriate segment selectors are established, but the segment may not actually be read into memory until it is first accessed. Such segments are termed "load on call" segments, and are the default code and data segment type. You can force the loader to physically load the segment when the program first begins running by assigning the segment the "preload" attribute in the module definition file. You can postpone physically loading a segment because of the virtual memory mechanism used in the protected mode (discussed in Chapter 5).

Supplying Addresses of Dynamic-Link Functions Once the system has loaded the dynamic-link library containing the function listed in the relocation record, it must go through the associated linked list, writing the address of this function to the address fields of the CALL instructions on the list. Since a dynamic-link function is always contained in a separate segment, it must be called with a far CALL instruction; the loader must therefore supply both the segment selector and the offset of the function.

The relocation record specifies the entry point either as an ordinal value or as a name. The ordinal value serves as an index into the entry table found in the header of the dynamic-link library file. If the entry point is specified as an ordinal value, the loader reads the corresponding entry in

this table, which identifies the segment containing the function and gives the offset of the function within this segment. The loader supplies the appropriate segment selector for the specified segment, which it has just loaded into memory, and uses the offset value obtained from the entry table. The resulting selector:offset address is then written to all CALL instructions on the linked list within the program.

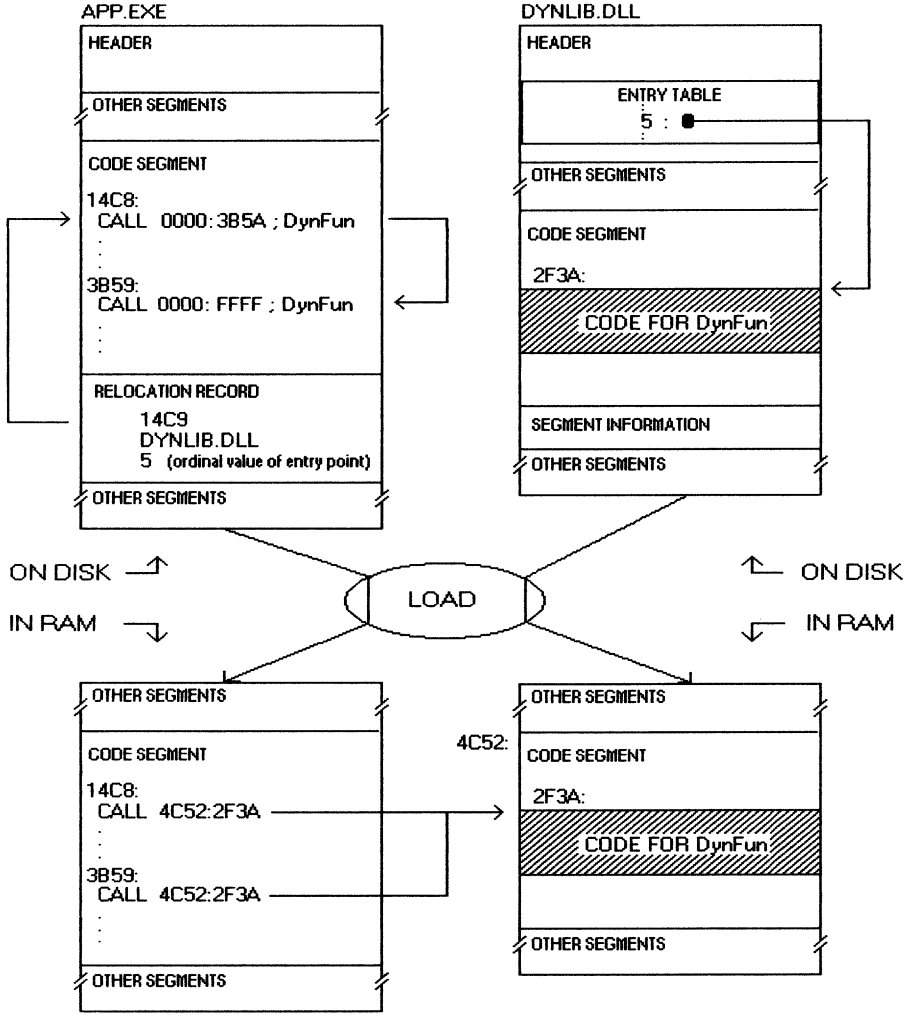
If the entry point of the dynamic-link function is identified by name, the loader must first look in a table of entry point names within the header of the dynamic-link library. This table supplies the ordinal value for each function that it lists. The loader must then obtain the function location from the entry table. Thus, identifying dynamic-link functions by entry point name involves an extra step. Chapter 4 will show that using names rather than ordinal values is not only slightly slower, but also consumes more memory.

Note that the loader follows a special procedure for certain system services—these services are termed resident functions. Although the programmer calls these functions in the same manner as normal dynamic-link routines, the loader resolves references to the functions by supplying the entry point of a routine within the operating system kernel and does not load a separate dynamic-link library file. These functions are described in the section on The Uses of Dynamic-linking, later in the chapter.

See Chapter 8 for more information on loading dynamic-link library segments—specifically, how the system sets up descriptors for these segments.

Figure 3-4 illustrates the process of loading a program that contains a call to a dynamic-link function (this figure is based on the same example shown in Figure 3-2, which shows how the program was processed by the linker). The example depicted in Figure 3-4 references the function entry point using an ordinal value, rather than an entry point name. Compare this example to Figure 3-3, which illustrates the loading of a program containing a statically linked function. Note that these figures isolate specific processes from the many processes that occur when a program is loaded. A typical OS/2 program contains many calls to both statically and dynamically linked functions.

Figure 3-4: Loading a program and a dynamic-link library.



Calling the Dynamic-Link Function

Once the program and all referenced dynamic-link libraries have been loaded, matters become simple. Calls to dynamic-link library functions become direct far calls to the dynamic-link code in memory. Although a dynamic-link function is contained in a separate disk file, it runs as part of the same process as the client program—precisely, it runs as part of the process thread from which it is called. Calling a dynamic-link function is similar to calling a normal subroutine within a large memory model C program. There are two important features of dynamic-link functions that you should consider as you begin developing dynamic-link libraries.

First, a dynamic-link function can be used by several simultaneous processes, namely the dynamic-link function can be called by more than one process at a given time. If the dynamic-link library employs only instance data segments, the existence of multiple client processes is largely masked. If, however, it uses one or more global data segments, or if it manages other shared objects, such as memory segments or devices, it must smoothly coordinate the activities of the separate processes. These issues are discussed in Chapter 5.

Second, like a normal program subroutine, a dynamic-link function has full access to all objects owned by the client process, such as memory segments, file handles, and semaphores. It can also allocate additional objects, which become owned by the common process. Accordingly, the dynamic-link function must be considerate in using allocated objects so that it will not sabotage the client program. For example, it should neither arbitrarily close file handles that were opened by the client program, nor open a large number of files without increasing the limit on the number of handles that can be opened by the process. Also, certain system calls, such as **DosSetPrty** which sets the priority of a thread or a process, can directly affect the client program and must be used with care.

Terminating the Program

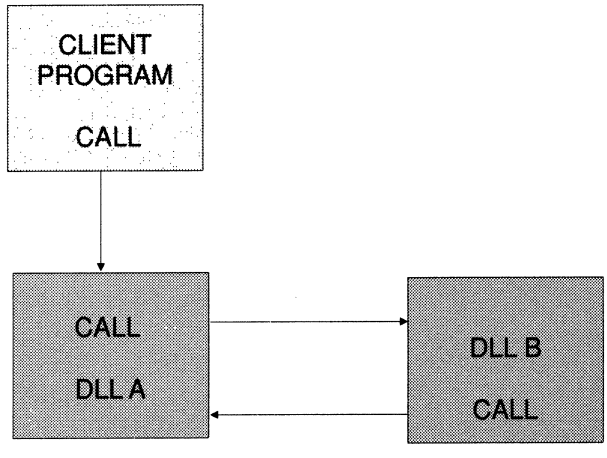
Before the program terminates, a package of dynamic-link routines may provide a function for the client program to call when it has completed using the package. A Presentation Manager application can call WinTerminate to sever its ties with the Presentation Manager and to release

any remaining objects maintained for the client program. As shown in Chapter 2, the Presentation Manager is implemented as a package of dynamic-link routines.

A dynamic-link package can also install a termination routine that is automatically called when the client process terminates—regardless of whether the termination is normal or through an error condition. Such a routine is useful if the client program fails to call the appropriate function to notify the dynamic-link package that it has completed using its services, or if the package does not provide such a routine. Chapter 6 describes how to write and install termination routines.

When the last client process using a given dynamic-link library terminates, the system frees the dynamic-link library from memory. Knowing when to release a dynamic-link library is not a matter of maintaining a reference count of client processes, because these references can be circular (remember that a dynamic-link library function can call a function in another dynamic-link library). For example, in Figure 3-5, the program references dynamic-link library A, which references dynamic-link library B, which in turn references dynamic-link library A.

Figure 3-5: Circular references among dynamic-link libraries.



If the program terminates, dynamic-link libraries A and B would still have one reference each; however, these libraries no longer serve a purpose. Accordingly, the system releases a dynamic-link library when it can no longer trace a path of references from this library back to a client program. Thus, libraries A and B would both be freed from memory.

The Uses of Dynamic Linking

This section describes how these functions fit into the overall layered structure of OS/2. The discussions emphasize the unique advantages offered by the dynamic linking mechanism for each type of use.

The first general use for the dynamic-linking mechanism is to provide a convenient method for application programs to obtain the basic services of the operating system kernel. These services are provided by the set of dynamic-link functions having the Dos prefix, which were demonstrated in Chapter 1.

Some of the operating system services, however, cannot be performed by a normal dynamic-link library. As you have seen, a dynamic-link function executes as part of the client process. Under OS/2, however, normal application processes have the lowest privilege level in the system. In the protected-mode of the 80286 and later model processors, programs run at one of four privilege levels—the privilege level determines which memory segments the process can access and which machine instructions it can execute. The operating system kernel operates at the highest level of privilege and can, therefore, access all segments in the system and use all available machine instructions. An application program, running at the lowest privilege level, is restricted in the segments it can access and the machine instructions it can execute. See Chapter 10 for more information on privilege levels.

The allocation of a memory segment requires the highest privilege level. Therefore, an operating system function that provides such a service cannot be performed by a normal dynamic-link library—which operates within an application process—rather, the service must be accomplished by a routine that is within the operating system kernel.

Accordingly, when the loader resolves references to certain system functions—known as **resident functions**—rather than reading a dynamic-link library into memory and obtaining the function entry address from this library, the loader assigns the CALL instruction the address of the appropriate routine within the operating system kernel. More precisely, the address assigned to the CALL instruction contains a segment selector for a **call gate**, which is a special segment descriptor that points to a code segment at a higher privilege level. Making a function call through a call gate allows a program to temporarily execute at a higher privilege level. This mechanism does not breach system protection, however, since only the system can set up call gates, and application programs are allowed to execute kernel code only through a highly restricted set of entry points.

The operating system maintains a list of the resident functions, which must be resolved in the manner described. This list can vary depending upon the version of OS/2; however, whether a function is executed by a dynamic-link library or by a kernel routine does not affect the function-calling protocol or the client program.

From the viewpoint of the programmer, the primary advantage of using the dynamic-linking mechanism to access the services of the operating system is that these services can be called in the same manner as normal external functions—using the standard calling protocol employed by high-level languages.

A second basic use for the dynamic-link mechanism is to provide access to the function subsystems supplied by the operating system. A subsystem is a collection of related dynamic-link functions, typically used to manage a device that can be shared by many processes. As you saw in Chapter 1, OS/2 provides subsystems for managing the keyboard, the mouse, and the screen (the Kbd, Mou, and Vio functions, respectively). The features of the dynamic-linking mechanism are ideally suited for supporting subsystems managing shared devices. Specifically, dynamic-link libraries can easily perform required device initializations when first loaded (through initialization routines, which are described in Chapter 6); through instance data segments, they can maintain separate information for each calling process and they can maintain information on the state of the device itself within a global data segment (described in Chapter 5).

Dynamic-link libraries are also useful in the implementation of major operating system extensions. Examples of operating system extensions that have been implemented as sets of dynamic-link libraries include the Presentation Manager and the communications and database facilities of the OS/2 Extended Edition developed by IBM. Because the dynamic-link mechanism is well documented and because adding dynamic-link libraries to the system does not entail modification of the system code, other software developers have an opportunity to develop similar operating system extensions.

A significant advantage of using the dynamic-link mechanism to extend the operating system is that such extensions integrate smoothly with the basic operating system services. An operating system extension can be installed by merely copying additional .DLL files onto the hard disk; its functions can be called in the same manner as those of the basic operating system. Dynamic linking provides an open pathway for expanding the facilities of OS/2.

Finally, dynamic-linking is useful for packaging collections of routines, which you can use for your own programs, or distribute as commercial function libraries. Whereas a function library for MS-DOS is typically distributed as a collection of object (.OBJ) or library (.LIB) files, a library for OS/2 could be distributed as a collection of dynamic-link library (.DLL) files. A package for either system may include the source code in addition to, or instead of, the binary code. When the final applications are shipped to the user, they must be accompanied by all referenced dynamic-link library files.

Packaging functions within dynamic-link libraries can save space both on the disk and in RAM. Although the user may run several programs that call these functions, only a single copy of the functions needs to be stored on the disk, and only a single copy of the code segments must be loaded into memory when the programs are run. Also, by following the guidelines discussed in this book (especially in Chapter 4), you can develop dynamic-link libraries that can be called from programs developed in any language that supports OS/2. Additionally, the user can install updated and enhanced versions of dynamic-link functions without the need to obtain new executable versions of the applications that call these functions—provided the calling protocol for the functions remains

the same. Subsequent versions of the operating system may provide enhanced versions of system services that offer higher performance. Programs that use these services automatically will begin using the latest function versions without the need to recompile or relink the applications.

In general, packaging software tools within dynamic-link libraries enhances the ideal of code and data abstraction. According to this ideal, the systems programmer who writes a set of functions hides the details of the implementation of the functions and the internal data structures from the applications programmer who uses the functions. The systems programmer, however, publishes the function-calling protocol and the use of any public abstract data types associated with these functions. Accordingly, the systems programmer can freely enhance the implementation of the functions as long as the public interface is left unaltered. Likewise, the applications programmer can freely use the functions without concern for their implementation, and without building into the application a dependency upon specific implementation details.

Some high-level languages, such as Ada and Modula2, provide greater intrinsic support for abstraction than C does. The dynamic linking mechanism, however, enforces a high level of independence between a C program and the library functions that it calls and therefore enhances the level of abstraction for programs written in any language. For example, changes in the implementation of a dynamic-link library are less likely to affect the calling program than similar changes in a statically-linked library. For example, increasing the code size of a dynamic-link library cannot force the calling program to adopt a larger memory model. In fact, as mentioned previously, enhanced versions of the functions can be supplied directly to the application user without even involving the applications programmer.

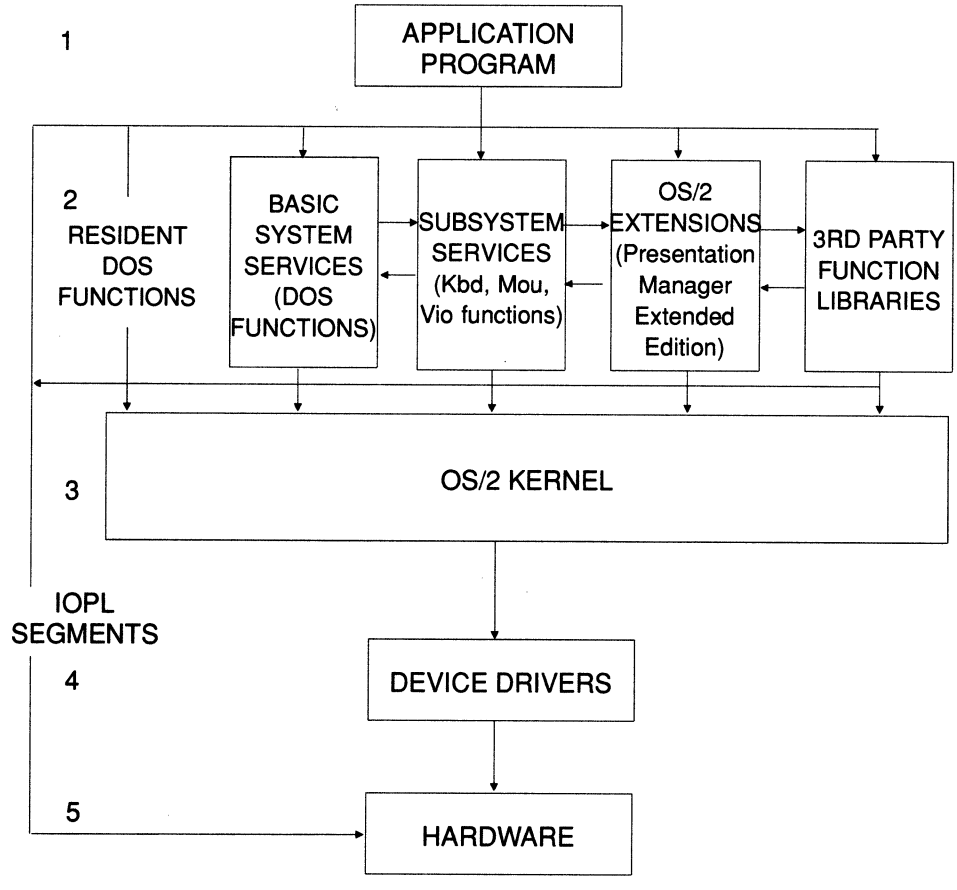
Dynamic-Link Libraries within the Structure of OS/2

Figure 3-6 illustrates the position occupied by various types of dynamic-link libraries within the layered structure of the OS/2 operating system. This figure depicts the following five basic operating system layers:

1. The application program level, including the C runtime library and other statically linked library functions.

2. The dynamic-link library level.
3. The operating system kernel level.
4. The device driver level.
5. The hardware level.

Figure 3-6: The layered architecture of OS/2.



In general, an application obtains a required service by making a call to a dynamic-link library function, and control passes through each layer of the operating system, eventually accessing an underlying hardware device. As you can see in Figure 3-6, there are some exceptions to this normal flow of control. First, an application or a dynamic-link function can directly control a hardware device through an I/O privileged code segment; Chapter 12 discusses how to write functions that execute with I/O privilege. Second, as described earlier in this section, calls to resident Dos functions directly invoke routines within the kernel rather than an actual dynamic-link function. Finally, dynamic-link functions can call functions in other dynamic-link libraries, often forming complex and circular flows of control.

CHAPTER 4

CREATING A DYNAMIC-LINK LIBRARY

In the first three chapters you have learned how to use the dynamic-link library functions provided by OS/2 and the Presentation Manager, and you have seen how the dynamic linking mechanism works. In this chapter, you will discover how to create a dynamic-link library.

The chapter is based upon an example dynamic-link library, which contains a set of functions for managing the printer and for printing formatted reports. These functions can be called from a standard OS/2 protected-mode program or from a Presentation Manager application.

The chapter explains and illustrates all of the basic steps required to develop and use a simple dynamic-link library. Methods for adding advanced features are presented in subsequent chapters. Specifically, a dynamic-link library developed using the techniques given in this chapter is subject to the following constraints:

- The dynamic-link library can use only instance data segments (that is, separate data segments are loaded for each client process). The techniques for sharing data within global data segments are discussed in Chapter 5.

- The dynamic-link library has neither an initialization nor a termination routine; adding initialization and termination routines is discussed in Chapter 6.
- The dynamic-link library is not free to call functions belonging to the standard C runtime library. Methods for using special versions of this library within a dynamic-link module are presented in Chapter 7.
- The example client program presented in this chapter uses only load-time dynamic linking; runtime dynamic linking is presented in Chapter 8.
- The dynamic-link code is developed in the C language, and does not contain I/O privileged routines (explained in Chapter 3); techniques for writing dynamic-link libraries in assembly language and for developing I/O privileged routines are presented in Chapter 12.

The chapter begins by giving an overview of the entire process of developing a dynamic-link library. It then presents the source code and supporting files for an example dynamic-link module, and explains in detail each step of its development. The chapter concludes by presenting a program that uses the functions in the example module.

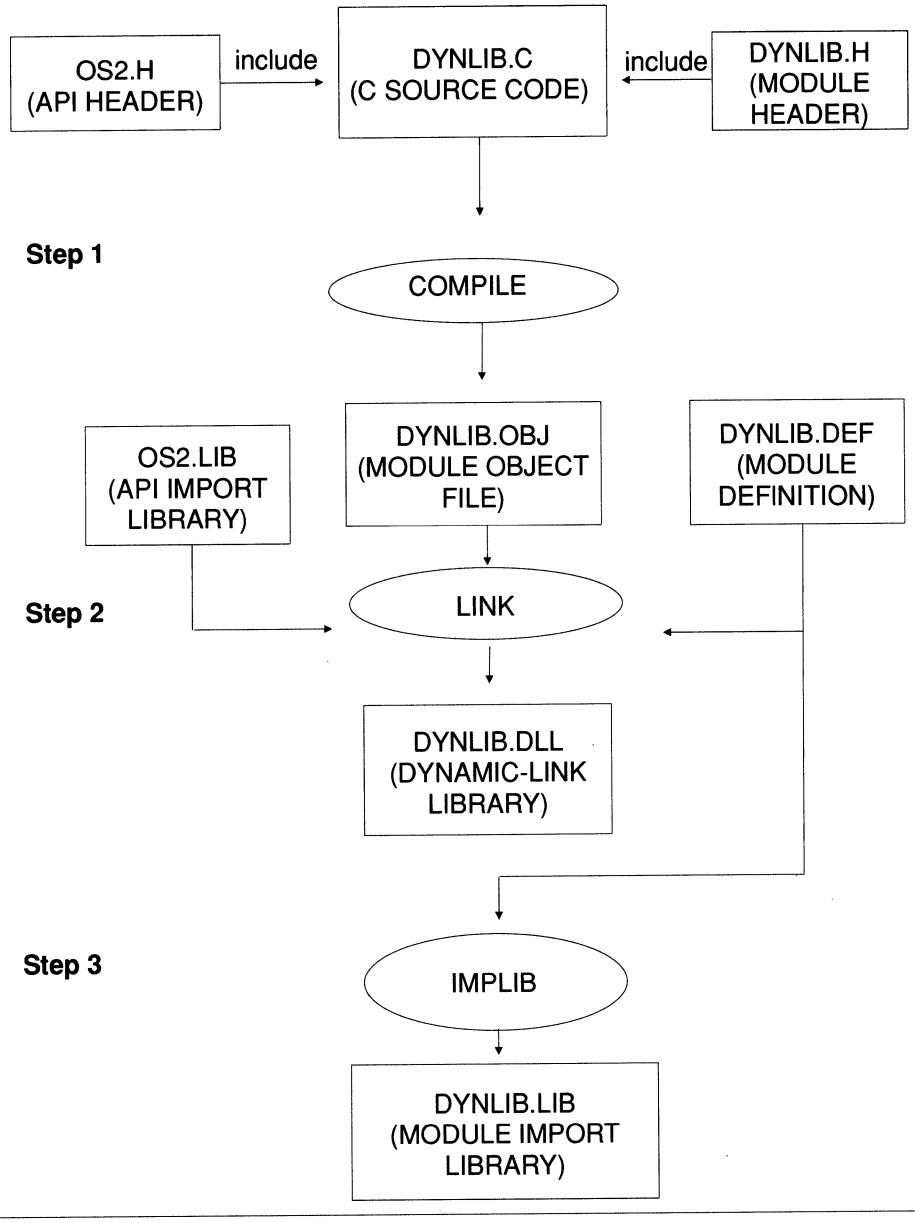
An Overview of the Process

Figure 4-1 illustrates the three basic steps for preparing a dynamic-link library module. These steps are as follows:

1. Compile the source file.
2. Generate the executable dynamic-link library using the linker.
3. Produce an import library.

This section provides a brief overview of these procedures; subsequent sections explore each step in detail.

Figure 4-1: The three basic steps for creating a dynamic-link library.



As you can see from Figure 4-1, the first two steps for preparing a dynamic-link library are the same as those required to create a normal executable program. The first step is to compile the source code. If the dynamic-link module calls OS/2 API functions (either the basic operating system functions or the services available for Presentation Manager applications) it must include the appropriate system header files and any header files containing information specific to the module. As you will see later in the chapter, declarations and definitions required by the calling client program are normally placed in a module header file, which is included in both the dynamic-link module and in the calling program. Note the absence in Figure 4-1 of the usual C library header files—as mentioned, the simple dynamic-link library presented in this chapter does not call C library functions.

If you are writing a larger dynamic-link module, you might want to place the code in more than one source file. In the same manner as a normal C application, these source files are compiled separately and the resulting object modules are combined during the linking phase. (Note that this book uses the term **module** in a general sense. It is used to refer to an object file or the contents of this file within a library file. It is also used to refer to a single dynamic-link library, which may have been created from one or more object modules.)

The second step is to use the linker to prepare the actual dynamic-link library file (.DLL file). A dynamic-link library is prepared from one or more object modules in the same manner as a normal executable program. In fact, the final format of a dynamic-link library closely resembles that of a protected-mode program. Both types of files contain the same header format, and the same layout of code and data segments with their accompanying relocation information (described in Chapter 3). A one-bit flag within the program header indicates whether the file is an executable program or a dynamic-link library (specifically, bit 15 of the flag word at offset 0Ch in the OS/2 portion of the header). As you will see later in the chapter, if the module definition file contains a NAME statement, the linker generates an executable program, but if it contains a LIBRARY statement, it generates a dynamic-link library.

If your dynamic-link module calls functions in other dynamic-link libraries—such as those containing the OS/2 API functions—you must

supply the linker with the names of the corresponding import library or libraries to resolve the references to these functions (such as OS2.LIB or DOSCALLS.LIB). Alternatively, you can list these functions in an IMPORTS statement in the module definition file, as described later in the chapter.

When linking the dynamic-link library, you must also supply a module definition file, which is optional for normal protected mode-programs, but is required for dynamic-link libraries. This file serves not only to tell the linker to generate a dynamic-link library, but is also required to specify the names of the functions that can be called by a client program, and to define other features of the resulting dynamic-link library.

The third step is to use the IMPLIB utility to generate an import library containing a dynamic-link record for each function in your module that can be called by a client program. The import library is used to resolve references to these functions when linking a client program (the use of import libraries is explained in Chapter 3). Note that creating an import library is optional, since a client program can also resolve references to dynamic-link functions by listing these functions in an IMPORTS statement in the module definition file.

An Example Dynamic-Link Library

The example dynamic-link module presented in this chapter consists of a set of functions for managing the printer and for printing text. These functions are especially useful for generating formatted reports. Using the functions, you can perform the following specific tasks:

- Determine whether the printer is ready to receive output (**PrtReady**).
- Initialize the printer (**PrtInit**).
- Send a single character or control code to the printer (**PrtPutC**).
- Send a NULL-terminated string of characters or control codes to the printer (**PrtPutS**).

- Print a string at a specified row and column position (**PrtPosition**).
- Start a new page, optionally generating a formfeed (**PrtNewPage**).

In general, you can use this dynamic-link module either from a standard protected-mode program or from a Presentation Manager application. However, there are several restrictions in the use of the functions. First, if the functions are called by more than one thread within a single process, the printed output will become interspersed on the page, since the module makes no provision for segregating the output generated by multiple threads. Also, if you have *not* installed the OS/2 print spooler, the same restriction would apply to calling the functions from more than one concurrent process or screen group. If the OS/2 print spooler is enabled, however, it automatically segregates the printer output produced by separate processes, and you may therefore call the functions from more than one simultaneous process. The spooler stores the output from each process in a spool file, which is printed when the application closes the printer file handle or terminates.

Another restriction is that you should not perform a lengthy operation, such as printing a report, from the main thread of a window procedure within a Presentation Manager application. A window procedure should return quickly (the documentation recommends returning control within 0.1 second) so that the program can continue processing messages; otherwise, the user cannot perform other tasks or switch Presentation Manager windows. Accordingly, you should start a separate program thread to perform the printing operations (using the methods discussed in Chapter 1), and allow the main thread to return immediately.

The C source code for the example dynamic-link module is listed in Figure 4-2. The module header file is given in Figure 4-3 which comes later in the Chapter. This file is included in the source listing of Figure 4-2 and must also be included within any program that calls the module functions. Figure 4-4 also comes a bit later and presents the module definition file required to link the dynamic-link library and, finally, Figure 4-5 provides a MAKE file for generating the dynamic-link library and corresponding import library.

Figure 4-2

```
/*
     Figure 4-2

     This source file defines a set of dynamic-link functions for managing the
     printer and for printing formatted reports.  It contains the following
     functions that may be called by a client program:

     PrtReady       Determines whether the printer is ready to receive output.
     PrtInit        Resets the printer.
     PrtPutC        Sends a single character to the printer.
     PrtPutS        Sends a NULL-terminated character string to the printer.
     PrtPosition    Prints a string at a specified row and column position.
     PrtNewPage     Generates a new page;  used in conjunction with PrtPosition.

     Preparing the dynamic-link library file from this source listing requires
     the following additional files:

          FIG4_3.H       Header file included in FIG4_2.C and in the client
                         program.
          FIG4_4.DEF     Module definition file.
          FIG4_5.MAK     MAKE script for preparing the program.
*/

#define INCL_DOS
#include <OS/2.H>
```

```
#include "FIG4_3.H"

#define PRTNAME "LPT1"          /* Printer device name.                    */

int _acrtused = 0;              /* Define variable to avoid linking in C   */
                                /* startup code.                           */

                                /* External variables for storing printer state*/
unsigned char Opened = 0;       /* Indicates whether printer has been opened.  */
HFILE Handle;                   /* Handle for printer device.              */
int CurRow = 1;                 /* Current row of printer head.            */
int CurCol = 1;                 /* Current column of printer head.         */

                                /* Private functions:                      */
USHORT _PrtOpen                 /* Opens the printer.                      */
     (void);
USHORT _StrLen                  /* Calculates length of a string.          */
     (char far *String);

unsigned pascal far _loadds PrtReady
     (unsigned char far *PtrFlagReady)
/*
     This function assigns a nonzero value to '*PtrFlagReady' if the printer
     is ready to receive output;  it assigns zero to '*PtrFlagReady' if the
     printer is not ready.  If successful, it returns zero;  if an error
     occurs, it returns a nonzero API error code.
*/
     {
     USHORT ErrorCode;              /* Stores the API error code.          */
     BYTE PrinterStatus;            /* Receives printer status.            */
```

```
    BYTE Reserved = 0;              /* Reserved DosDevIOCtl parameter.        */
    if (!Opened)                    /* Open printer if necessary.             */
         {
         ErrorCode = _PrtOpen ();
         if (ErrorCode)
              return (ErrorCode);
         }

    ErrorCode = DosDevIOCtl  /* Send I/O control command to printer driver. */
         (&PrinterStatus,    /* Receives printer status.                    */
         &Reserved,          /* Reserved:  must point to 0 variable.        */
         0x0066,             /* GETPRINTERSTATUS function.                  */
         0x0005,             /* Function category.                          */
         Handle);            /* Printer device handle.                      */
    if (ErrorCode)
         return (ErrorCode);

    *PtrFlagReady = PrinterStatus & 0x10;   /* Mask all bits except        */
                                            /* 'printer selected' bit.     */
    return (0);

    } /* end PrtReady */


unsigned pascal far _loadds PrtInit
     (void)
/*
     This function initializes the printer device.  If successful, it returns
     zero;  if an error occurs, it returns a nonzero API error code.
```

```
*/
     {
     USHORT ErrorCode;              /* Stores the API error code.            */
     BYTE Reserved = 0;             /* Reserved DosDevIOCtl parameter.       */

     if (!Opened)                   /* Open printer if necessary.            */
          {
          ErrorCode = _PrtOpen ();
          if (ErrorCode)
               return (ErrorCode);
          }

     ErrorCode = DosDevIOCtl  /* Send I/O control command to printer driver. */
          (0L,                /* Must be 0 for this function.                */
          &Reserved,          /* Reserved:  must point to 0 variable.        */
          0x0046,             /* INITPRINTER function.                       */
          0x0005,             /* Function category.                          */
          Handle);            /* Device handle.                              */
     if (ErrorCode)
          return (ErrorCode);

     return (0);

     } /* PrtInit */


unsigned pascal far _loadds PrtPutC
     (int Ch)
```

```
/*
     This function sends character 'Ch' to the printer.  If successful, it
     returns zero;  if an error occurs, it returns a nonzero API error code.

     Warning:  Neither 'PrtPutC' nor 'PrtPutS' should not be used in conjuntion
     with 'PrtPosition' unless the function is used to send a control code that
     does NOT move the printer head (otherwise the internal record of the
     current printer row and column maintained by the module would become
     invalid).
*/
     {
     USHORT ErrorCode;              /* Stores the API error code.            */
     USHORT BytesWritten;           /* Number of bytes successfully printed. */

     if (!Opened)                   /* Open printer if necessary.            */
          {
          ErrorCode = _PrtOpen ();
          if (ErrorCode)
               return (ErrorCode);
          }

     ErrorCode = DosWrite                    /* Send character to printer.   */
          (Handle,                           /* Printer device handle.       */
          &Ch,                               /* Address of char. to print.   */
          1,                                 /* Number of bytes to print.    */
          &BytesWritten);                    /* Assigned bytes written.      */
     if (ErrorCode)
          return (ErrorCode);


     return (0);
```

```
    } /* end PrtPutC */
unsigned pascal far _loadds PrtPutS
    (char far *String)
/*
    This function sends the NULL terminated string 'String' to the printer.
    See the warnings given for 'PrtPutC', which apply to 'PrtPutS' as well.
    If successful, it returns zero;  if an error occurs, it returns a
    nonzero API error code.
*/
    {
    USHORT ErrorCode;              /* Stores the API error code.             */
    USHORT BytesWritten;           /* Number of bytes successfully printed.  */

    if (!Opened)                   /* Open printer if necessary.             */
        {
        ErrorCode = _PrtOpen ();
        if (ErrorCode)
            return (ErrorCode);
        }

    ErrorCode = DosWrite                    /* Send string to printer.       */
        (Handle,                            /* Printer device handle.        */
        String,                             /* Address of string to print.   */
        _StrLen (String),                   /* Number of bytes to print.     */
        &BytesWritten);                     /* Assigned bytes written.       */
    if (ErrorCode)
        return (ErrorCode);

    return (0);
```

```
    } /* end PrtPutS */
unsigned pascal far _loadds PrtPosition
    (char far *String,
    int Row,
    int Col)
/*
    This function prints NULL terminated 'String' beginning at the position
    specified by 'Row' and 'Column'.  If successful, it returns zero;  if an
    error occurs, it returns one of the following error codes:

        BADPOSITION         The requested print position was to the left of
                            or above the current printer head position.

        For all other errors, it returns the API error code.

    The following rules must be observed:

    o  The string must NOT contain control characters (i.e., any characters
       that do not advance the print head a single column).  To send control
       codes, use 'PrtPutC' or 'PrtPutS'.
    o  The string must not contain tab characters.
    o  The string must not contain newline characters.  To advance to a new
       line, use a subsequent call specifying the desired row.  Do not send
       more characters than can fit on the current line.
    o  To generate a new page and reset the row and column numbers, use
       'PrtNewPage'.  Do not send more lines than can fit on a single page.
    o  'PrtPosition' and 'PrtNewPage' should be used by only a single thread
        within a process at a given time.
*/
    {
```

```
USHORT ErrorCode;              /* Stores the API error code.              */
/*** Test for valid row and column. ************************************/
if (Row   CurRow ||
    Row == CurRow &&
    Col   CurCol)
     return (BADPOSITION);
/*** Print CR/LF pairs until reaching desired row. **********************/
while (Row - CurRow)
     {
     ErrorCode = PrtPutS ("\x0d\x0a");
     if (ErrorCode)
          return (ErrorCode);
     ++CurRow;                 /* Adjust record of printer position.      */
     CurCol = 1;
     }
/*** Print spaces until reaching desired column. ************************/
while (Col - CurCol)
     {
     ErrorCode = PrtPutC (32);
     if (ErrorCode)
          return (ErrorCode);
     ++CurCol;                 /* Adjust record of current column.        */
     }
/*** Print the string. *************************************************/
ErrorCode = PrtPutS (String);
if (ErrorCode)
```

```
        return (ErrorCode);
    CurCol += _StrLen (String);    /* Adjust record of current column.        */

    return (0);

    } /* end PrtPosition */

unsigned pascal far _loadds PrtNewPage
    (unsigned FlagFormFeed)
/*
    This function resets the internal row and column counters for the
    position of the printer head;  if 'FlagFormFeed' is nonzero, the
    function also generates a formfeed.  If successful, it returns zero;
    if an error occurs, it returns a nonzero API error code.
*/
    {
    USHORT ErrorCode;              /* Stores the API error code.              */

    if (FlagFormFeed)
        {
        /*** Generate a carriage return and form feed. *********************/
        ErrorCode = PrtPutS ("\x0d\x0c");
        if (ErrorCode)
            return (ErrorCode);
        }

    /*** Reset current row and column. ************************************/
    CurRow = CurCol = 1;
```

```
    return (0);
    } /* end PrtNewPage */

/*** Private functions: ****************************************************/

USHORT unsigned _PrtOpen
    (void)
/*
    This private function opens the printer device.  If successful, it returns
    zero;  if an error occurs, it returns a nonzero API error code.
*/
    {
    USHORT ErrorCode;              /* Stores the API error code.            */
    USHORT Action;                 /* Receives 'DosOpen' action code.       */

    ErrorCode = DosOpen
        (PRTNAME,             /* Device name for printer.                   */
        &Handle,              /* Receives printer device handle.            */
        &Action,              /* Receives action code.                      */
        0L,                   /* Initial allocation size:  n/a.             */
        0,                    /* File attribute:  n/a.                      */
        1,                    /* Open flag:  open file if it exists.        */
        0x0041,               /* Open mode:  write access and share.        */
        0L);                  /* Reserved:  must be 0.                      */
    if (ErrorCode)
        return (ErrorCode);

    Opened = 1;               /* Set opened flag.                           */

    return (0);

    } /* end _PrtOpen */
```

```
USHORT _StrLen
     (char far *String)
/*
     This private function returns the length of the NULL-terminated string
     'String'.
*/
     {
     USHORT Count = 0;

     while (*String++)
          ++Count;

     return (Count);

     } /* end _StrLen */
```

An example application that uses the functions in this module is presented at the end of the chapter (in Figure 4-7).

Writing the C Source Code

This section describes the most important step in creating a dynamic-link module: writing the source code. The discussion is based upon the example module of Figure 4-2. It begins by describing the general guidelines for writing a dynamic-link module, and then explains the implementation of each of the functions in the example module.

General Guidelines This section describes some general guidelines for writing dynamic-link libraries in C; most of these guidelines are illustrated by the example dynamic-link module of Figure 4-2.

Figure 4-3

```
/*
     Figure 4-3

     This header file contains the public declarations and definitions for the
     dynamic link module of Figure 4-2.  It is included in the dynamic-link
     source code (FIG4_2.C) and should be included in any client program that
     calls one or more of these functions.
*/

unsigned pascal far _loadds PrtReady
     (unsigned char far *PtrFlagReady);

unsigned pascal far _loadds PrtInit
     (void);

unsigned pascal far _loadds PrtPutC
     (int Ch);

unsigned pascal far _loadds PrtPutS
     (char far *String);

unsigned pascal far _loadds PrtPosition
     (char far *String,
     int Row,
     int Col);

unsigned pascal far _loadds PrtNewPage
     (unsigned FlagFormFeed);

                                   /* Prt module error constant:            */
#define BADPOSITION    1000        /* 'PrtPosition': bad position passed.   */
```

The first guideline is that the program should define the variable **_acrtused** to prevent the linker from binding in the C startup code. The C startup code is required only for a standard application program (or for a dynamic-link library that uses one of the special versions of the C runtime library described in Chapter 7); it is not required for a dynamic-link library that does not use the C runtime library. The compiler automatically writes the name _acrtused to the object file as an external reference. Since this variable is defined within the C startup code, which is located in the C runtime library file, its presence as an external reference in the program object file causes the linker to bind the startup code into the program. By defining this variable in the C source file, the reference is resolved immediately and the startup code is eliminated. As you will see, the C library is not even included on the linker command line. The example module defines this variable equal to 0, although you can initialize it to any value.

Although dynamic-link libraries written according to the methods that are described in this chapter cannot call C library functions, they are free to call many of the functions belonging to the OS/2 API. If a dynamic-link module is designed to be used by either a standard protected-mode program or by a Presentation Manager application, then it should call only the basic Dos functions. If the module is designed to be called only by a Presentation Manager application, then it can call the Dos functions, as well as the Win, Gpi, and other Presentation-Manager-specific services. If the module is designed to be used only by non-Presentation-Manager programs, it can call any of the Dos, Kbd, Mou, and basic Vio functions. (These function categories are described in Chapters 1 and 2.)

In general, dynamic-link modules can freely call functions belonging to other dynamic-link modules. OS/2 can load a set of interdependent dynamic-link libraries into memory, which can call each other in complex patterns. As mentioned in Chapter 3, the references among these modules can even be circular. For example, module A calls module B, B calls C, and C calls A.

Unlike a normal subroutine called by a program in a single-tasking system, a dynamic-link function may be called by more than one concurrent thread within a single process, or by threads in more than one concurrent process. Each time a thread calls a dynamic-link function, it

creates an **instance** of this function (separate instances share the function code, but may or may not share the function data). A potential problem arises when separate instances of the function attempt to access the same memory location, device, or other shared object at the same time. For example, if two simultaneous instances of a function attempt to modify a memory variable at the same time, the resulting value may become invalid.

To prevent conflicts when accessing memory variables within a dynamic-link function, you should know how variables are shared among separate function instances. The following table, Table 4-1, lists three basic categories of data that can be declared within a dynamic-link library written in C:

Table 4-1: Data Categories

DATA CATEGORY	DESCRIPTION
automatic	Defined within the scope of a function, *not* using the static or external keywords; these variables are stored within the function stack frame or within machine registers.
non-shared permanent	Defined outside the scope of a function, or within a function using the static keyword, and located in an instance data segment.
shared permanent	The same as non-shared external data, but located in a global data segment.

Note that an automatic data item must be reassigned a value each time the function is called, and its value is lost as soon as the function exits. In contrast, a data item in one of the other two categories can be initialized when the program begins running, and retains its value throughout the life of the program unless it is explicitly reassigned—(hence the designation "permanent").

The dynamic-link module discussed in this chapter employs only the first two data categories; using shared external data is discussed in

Chapter 5. Table 4-2 shows how data in each of these categories is shared among separate function instances.

For example, the function PrtReady (defined in Figure 4-2) defines the following automatic data item:

```
BYTE PrinterStatus;
```

Each function instance would have a *separate* copy of this variable; therefore, there is no possibility of conflict among simultaneous instances that share the function code. For example, after one instance has stored a value in this variable, a second instance cannot alter the value.

Table 4-2: Data Sharing

DATA CATEGORY	SHARING OF A DATA ITEM
automatic	No sharing among function instances; a separate copy is created for each function instance.
non-shared permanent	Shared by all instances running within a single process; a separate copy is created for each separate client process.
shared permanent	All instances share the same copy.

In contrast, the module of Figure 4-2 defines the following variable outside the scope of a function:

```
int CurRow = 1;
```

This variable stores the current vertical position of the printer head, and is defined externally so that it can be accessed by any function in the module, and so that it retains its value between function invocations. Since the module uses an instance data segment, this variable is in the non-shared permanent category. Accordingly, each client process will have

its own copy of the variable, and separate clients will not sabotage each other's record of the printer head position.

However, if two simultaneous threads within the same process call this function, both instances will *share* the same copy of *CurRow*; overwriting each other's record of the printer head position. Because of this feature, and several others that will be discussed shortly, the example module is *not* suitable for simultaneous use by multiple threads within a single process, unless the process explicitly coordinates the activities of its separate threads.

In general, when function instances share a data item, you can prevent simultaneous access to this item by protecting the blocks of code that access the item with a semaphore or simple form of interprocess communication (the use of semaphores was discussed in Chapter 1). For example, if the variable *Count* is shared by separate function instances, the following code would prevent this variable from being accessed simultaneously:

```
DosSemRequest (&Sem,-1L);
if (Count= <60)
     ++Count;
else
     Count = 1;
DosSemClear (&Sem);
```

Note that there is no simple mechanism for a C program under OS/2 to obtain a separate copy of a variable for each function instance within a single process, which *maintains its value between function invocations.* An automatic variable is private to each instance, but loses its value between calls.

For example, if the module of Figure 4-2 supported calls from multiple program threads, it would have to maintain a separate record of the printer head position for each program thread. This could be achieved by dynamically allocating a block of memory to store data for each thread. You can obtain the identity of the calling thread through the OS/2 function **DosGetInfoSeg** (specifically, the identifier of the current thread is stored

in the **tidCurrent** field of the local information segment accessed through DosGetInfoSeg).

Dynamic-link modules that manage a given device must also coordinate the activities of simultaneous function instances to prevent conflicts when accessing this device. For instance, the example module given in this chapter manages the printer. As mentioned previously, the OS/2 print spooler automatically segregates printer output from separate processes. It collects printer output in individual files, and then prints these files one at a time.

If, however, two or more threads within a single application program simultaneously call the functions in the example module, the resulting printer output becomes interspersed on the page. Also, if one thread resets the printer by calling **PrtInit**, it will flush any output generated by other threads still stored in the printer's buffer.

Since the module has not been designed to separate the output produced by distinct threads, a multiple-thread client program would have to coordinate the activities of its own threads, and attempt to produce only one printed report at a time. To allow multiple program threads to simultaneously generate separate reports, the dynamic-link module has to store an individual record of the printer head position for each thread, collect printer output from each thread in a distinct temporary file (or memory buffer), and then print these files (or buffers) one at a time. The module would have to provide the essential features of a print spooler.

The next general technique is the method for defining functions. A dynamic-link module contains two basic types of functions: **private functions**, that are called only by other functions within the same module (within the current source file or within separate source files used to define your library), and **public functions**, that are called by a client program (and can also be called by other functions within the module).

For efficiency, the example module defines these two categories of functions differently. Private functions are defined as normal C functions, without using special keywords; for example the function **_PrtOpen** is called only within the module, and its definition begins as follows:

```
USHORT unsigned _PrtOpen
(void)
```

```
.
.
.
```

Definitions for public functions, however, use several special keywords. The definition of the first public function within the module begins as follows:

```
unsigned pascal far _loadds PrtReady
(unsigned char far *PtrFlagReady)
.
.
.
```

The special keywords used in this definition are an extension to the C language provided by the Microsoft C compiler and are not part of the standard ANSI definition. These keywords and their effects are listed in Table 4-3.

Table 4-3: Keywords and Effects

KEYWORD	EFFECT
pascal	(1) When the function is called, parameters are pushed on the stack in the *same* order as they appear in the parameter list. (2) The function removes the parameters from the stack (using the RET *n* machine instruction). (3) The number of parameters must be constant (a result of the first two features). (4) The function name is written to the object file in all uppercase letters, *without* adding a leading underscore.
far	**As a function type**: causes the compiler to call the function with a far call and causes the function to return with a far return instruction.

Table 4-3: Keywords and Effects

KEYWORD	EFFECT
	As a parameter type: the address passed as the parameter contains both an offset and a segment selector.
_loadds	The function loads the DS register with the selector of the data segment belonging to the dynamic-link module; it also restores the original value of DS immediately before it returns.

When declaring a public dynamic-link function, the **pascal** keyword is optional. The example module uses this option because it results in slightly more efficient code (executing the RET *n* instruction is faster than adjusting the stack pointer on function return), and it maintains uniformity with the OS/2 API functions, which are all declared as pascal functions.

The far keyword is required when defining a public function because the dynamic-link function code resides in a *different* segment than the client program code. You should *not* specify a calling type (either far or **near**) when defining a private function. Rather, private functions should use the default calling conventions for the current memory model (most dynamic-link modules use the small model; the resulting near calls are faster than far calls).

The far type modifier should also be used for declaring all address parameters passed to public functions, since the dynamic-link module has a separate data segment from that used by the client program.

Because the dynamic-link module uses its own default data segment, known as the automatic data segment, you must also declare all public functions with the **_loadds** keyword. A C function normally assumes that the DS register already contains the appropriate segment selector for the C automatic data segment, which is usually set by the C startup code. When a dynamic-link function is called from a client program however, the DS register contains the selector for the *client's* data segment. The _loadds keyword forces the function explicitly to load the appropriate

value into the DS register at function entry, and to restore the original value immediately before the function returns. An alternative method—the **/Au** compiler flag—will be explained later in the chapter. Note that private functions do not require this keyword since the DS register has already been properly set by a public function when the private function is called. A public function is always called before a private function receives control.

The dynamic-link module presented in this chapter is designed to be called from a C program (an example C program that uses the module is given in the last section). If you are writing a general purpose dynamic-link module, which may be called from a variety of programming languages, you can also include the **_saveregs** keyword when defining the public functions. This keyword forces the compiler to generate code within the function that saves and restores *all* of the machine registers—except registers AX and DX if they are used to return a value. A function declared with the _saveregs keyword can safely be called from languages that use other register saving conventions than those employed by Microsoft C. In Microsoft C it is necessary to save only the DS, SS, SP, BP, SI, and DI registers, in addition to the CS and IP registers, which are automatically restored when the function returns. As an example, the following function would save and restore all registers:

```
unsigned pascal far _saveregs _loadds DynaFun (void)
.
.
.
```

As you will see in the section on Writing the Supporting Files, declarations for all public functions should be placed in the module header file, which is included in both the module and the client source code. Declarations for private functions, however, should be placed at the beginning of the dynamic-link module source file. In the spirit of code abstraction, these functions should be kept hidden from the client process.

Another important guideline to consider when writing a dynamic-link library is to avoid creating side effects that impact the client process. As

described in Chapter 3, a dynamic-link function runs as part of the client process, and therefore has complete access to all memory segments, file handles, and other objects owned by the client. Also, any objects that the dynamic-link module allocates will be owned in common with the current client process, and will survive only until this client terminates. If the client does not explicitly deallocate these items, the system automatically deallocates them when the process ends.

A dynamic-link module, however, may require a stable set of memory segments, file handles, or other objects, which it can allocate and maintain independently of a client process, and which it can keep until the last client process terminates. To accomplish this, it can start a *separate process* (by calling the **DosExecPgm** API function), which can allocate all required items; the dynamic-link functions can then access these objects through an appropriate form of interprocess communication. Note, however, that the new process should be started as a detached (background) process. Otherwise, a client process, or an ancestor of a client process, could terminate the new process by calling **DosKillProcess**. Also, an ancestor process that calls **DosCWait** may be forced to wait until the new process terminates. You can find an explanation of these functions in the programmer's reference, and a general description of OS/2 processes in one of the books on basic OS/2 programming cited in the Bibliography.

As mentioned previously, it may be convenient to place the source code for a dynamic-link library in more than one source file. These files are compiled separately and are combined during linking in the same manner as an application program.

This section concludes with a discussion of several of the conventions used by the example dynamic-link module. First, following the convention employed by the OS/2 API, all public functions are named with a descriptive-three letter prefix (**Prt**). Ideally, the prefix used for a collection of related dynamic-link functions should be meaningful, and it should not conflict with a name already used by OS/2. For example, a set of screen management functions could have the prefix **Scr**, and the module should *not* use the Vio prefix.

Also in keeping with a convention used by the non-Presentation-Manager API functions, the values returned by the functions in the

example module are reserved for supplying the error status. If a function is successful, it returns 0; if an error occurs, it returns a nonzero code for the specific error. A value other than the error status is supplied to the calling program by assigning it to a program variable, the address of which is passed as a parameter.

The Functions This section briefly describes each of the functions in the dynamic-link module of Figure 4-2. See the source code listing for details on the calling protocols and values returned by each of these functions, and for additional information on their implementation.

PrtReady The function PrtReady reports whether the printer is currently ready to receive output.

PrtReady begins by checking the external flag *Opened* to determine whether the printer has been opened for the current process. Each function in the module that requires a valid handle to the printer device begins by checking this flag. If the flag is zero, the function calls the private subroutine _PrtOpen, which opens the printer, by calling the **DosOpen** function, and assigns the printer handle to the external variable, *Handle*. Note that *Opened* is initialized to zero and is set to 1 by _PrtOpen. Since the variables Opened and *Handle* are located in an instance data segment, separate copies exist for each new client process. This arrangement is appropriate since the printer must be opened for each process; a device handle is valid only within the process that opened the device.

Note that opening the printer is a task that could be performed by an initialization routine; specifically, an initialization routine that is called for each new client. In this case the module functions would not have to check the *Opened* flag each time they are called. Initialization routines, however, are not introduced until Chapter 6.

PrtReady obtains the current status of the printer by calling the **DosDevIOCtl** API function. This function is used to send control commands directly to device drivers. The specific control command sent by PrtReady causes the printer device driver to return a byte containing the current status of the printer. Since bit number 4 of this byte indicates whether the printer is *selected*, ready to receive output, all other bits in the status byte are masked to 0, and the resulting value is assigned to the receiving variable; the address of which is passed as the function

parameter. This value will be nonzero only if the printer is ready for output.

Note that you can use DosDevIOCtl to send a wide variety of control commands to various device drivers. The control commands that can be sent to the standard OS/2 device drivers are documented in the programmer's reference.

PrtInit The PrtInit function resets the printer. Resetting the printer clears any software control commands that have been sent, restores all of the printer's default settings, sets the printer's line counter to the top-of-form position, and flushes any data waiting in the printer's internal buffer. This function is useful for bringing the printer to a known state before beginning a new printing job.

PrtInit initializes the printer by sending the appropriate control command to the printer device driver through the DosDevIOCtl function—explained in the previous section.

PrtPutC PrtPutC prints a single character at the current position of the printer head. As you will see in the explanation of PrtPosition, you should not call this function in conjunction with PrtPosition or PrtNewPage, unless it is used to send a control code that does not move the printer head.

PrtPutC prints the character by calling DosWrite, which is the basic OS/2 function for writing to a file or device.

PrtPutS PrtPutS prints a NULL-terminated string at the current position of the printer head. Like PrtPutC, you should not call this function in conjunction with PrtPosition or PrtNewPage, unless you are sending a string of control codes that do not move the printer head.

PrtPutS sends the entire string to the printer with a single call to DosWrite. Note that DosWrite must be passed the number of characters to write; this value is calculated using the private function **_StrLen** (equivalent to the similarly named C library function, which unfortunately cannot be called from this module).

PrtPosition PrtPosition prints a NULL-terminated string beginning at the specified row and column position on the printed page. As you will see in the example program (in Figure 4-7, described later in the chapter), this function is useful for printing formatted reports, or for printing data

onto preprinted forms. Since PrtPosition maintains internal counters of the current row and column positions of the printer head, printing data using another function (such as PrtPutC or PrtPutS) would render these counters invalid. See the comments in the definition of the function (Figure 4-2) for additional rules.

PrtPosition moves the printer head to the specified starting position, and then prints the string, using the PrtPutC and PrtPutS functions.

PrtNewPage You should call this function before printing a new page with PrtPosition. It resets the row and column counters that store the current position of the printer head. They are both set to 1; therefore, the printer head should be at the top of the page when you call this function).

If the parameter **FormFeed** is nonzero, PrtNewPage also generates a carriage return and a form feed to eject the current page from the printer and to position the printer head at the beginning of a new page. When you call this function before printing the first page of a report, you can set PrtNewPage to 0 to avoid wasting a page.

Writing the Supporting Files

This section describes the two supporting files required to build the example dynamic-link module of Figure 4-2: the **module header file** and the **module definition file**.

The Module Header File The module header file is listed in Figure 4-3. This file is designed to be included within any client program that calls one or more of the module functions. In general, a module header file should contain the following items:

- Full prototype declarations for all functions that can be called by a client program (that is, the public dynamic-link functions).
- Definitions of any data types used in conjunction with these functions (for example, the definition of a structure used to exchange information with a public function).
- Symbolic constant definitions for values passed to or returned by the functions (such as constants for error return codes).

- Any other definitions or declarations that can simplify using the module (such as the macros provided by the OS/2 header files).

You can find examples of each of these items in the module header files that accompany the OS/2 API.

The module header file should be included not only in the client program but also in the module source file itself for two reasons. First, module functions may call other module functions, or use the constant or type definitions found in the header file. Second, if you include the function prototype in the header file, within the source file that defines the function, the compiler will automatically check the consistency between the two items. Accordingly, if you change the type or parameters of a function in the source file and forget to update the header file, the compiler will issue a warning.

The module header file serves as the definition of the module's public interface. You should therefore not define or declare items that are used only internally within the module such as the private functions in the example module, _PrtOpen and _StrLen. See the discussion on code and data abstraction in Chapter 3, in the section on The Uses of Dynamic-Linking.

Note that for statically linked functions—such as those in the C runtime library—the header file is frequently used to define shared variables. An example being a global error status variable that is assigned a value by the library function and is read by the calling application program. However, creating a variable that is directly shared between a dynamic-link function and its client programs is difficult, and is contrary to the principle of data abstraction. Such variables would have to be defined as external data, and would have to be exported by the dynamic-link module and imported by the client programs.

There are, however, two alternative methods for sharing an external variable that preserve the independence between the dynamic-link module and its client programs. First, you can share an external variable by explicitly exchanging its address. For example, the dynamic-link module could store the error status in a variable within its own data segment, and provide the **address** of this variable to the client through a function call. The following code example illustrates how the client process could obtain and store the address of such an error variable:

```
int far *GetErrorStatusAddr (void);          /* Dynamic-link function.*/
.
.
.
int far *PtrErrorStatus;
.
.
.
PtrErrorStatus = GetErrorStatusAddr ();
```

The client could subsequently obtain the current error status at any time through the expression ***PtrErrorStatus**.

Alternatively, the module could provide the error status or other value by returning this value directly from a function call. You could then provide a macro to simplify calling this function. For example, if the dynamic-link function

```
int GetErrorStatus (void);
```

returned the module error status, you could define the following macro within the module header file

```
#define PrtError   GetErrorStatus ()
```

The client program could then obtain the error status as if it were simply reading a global variable, as shown in the following example:

```
                                             /* Call a Prt function*/
if (PrtError)                                /* then call error handler*/
```

The Module Definition File The module definition file used for linking the example dynamic-link library is listed in Figure 4-4. This file contains two new statements that you have not seen in previous chapters. (Chapter 1 presented a module definition file for linking a general protected-mode program, and Chapter 2 gave a module definition file for linking a Presentation Manager application.) First, the file begins with the following statement:

```
LIBRARY FIG4_2
```

Figure 4-4

```
;               Figure 4-4
;
;               Module definition file for the dynamic-link module of Figure
                4-2.
;
LIBRARY FIG4_2

PROTMODE

DATA MULTIPLE

EXPORTS
                PRTREADY            @1
                PRTINIT             @2
                PRTPUTC             @3
                PRTPUTS             @4
                PRTPOSITION         @5
                PRTNEWPAGE          @6
```

The presence of a LIBRARY statement indicates that the linker should generate a dynamic-link library rather than an executable application. This statement also specifies the name of the module; this name should not include the .DLL extension, but should match the output filename given on the linker command line. The LIBRARY statement in Figure 4-4 specifies the filename FIG4_2, which matches the output filename given on the linker command line in the MAKE file of Figure 4-5—namely, FIG4_2.DLL. The MAKE file is described later in the chapter.

Figure 4-5

```
#    Figure 4-5
#    This MAKE file prepares the dynamic-link module of Figure 4-2.
#    The following files are involved:
#         FIG4_2.C
#         FIG4_3.H
#         FIG4_4.DEF
#
FIG4_2.OBJ : FIG4_2.C FIG4_3.H
     cl /c /W2 /ASw /G2s /Zp FIG4_2.C

FIG4_2.DLL : FIG4_2.OBJ FIG4_4.DEF
     link /NOI /NOD FIG4_2.OBJ, FIG4_2.DLL, NUL, OS2.LIB, FIG4_4.DEF

FIG4_2.LIB : FIG4_4.DEF
     implib FIG4_2.LIB FIG4_4.DEF
```

If a module definition file contains a LIBRARY statement, it should *not* contain a NAME statement, which indicates that the output file is to be an application. The NAME statement was used in the module definition

files in Chapters 1 and 2. If neither statement appears, the linker generates an application by default; therefore the LIBRARY statement is mandatory for creating dynamic-link libraries. The module definition file also contains the following new command:

```
DATA MULTIPLE
```

This statement causes the C automatic data segment to become an **instance data segment**. As you have seen, an instance data segment is one that is not shared among client processes; each client process has its own private copy of the segment, which must therefore be loaded from the disk every time a new client program is run. Note that this statement is equivalent to the following statement:

```
DATA NONSHARED
```

Chapter 5 describes the differences between instance and global data segments, and explains how to create and manage global segments. Also, the module definition file contains the following EXPORTS statement:

```
EXPORTS
  PRTREADY      @1
  PRTINIT       @2
  PRTPUTC       @3
  PRTPUTS       @4
  PRTPOSITION   @5
  PRTNEWPAGE    @6
```

As you saw in Chapter 2, when linking a Presentation Manager application, the EXPORTS statement must list the names of all window procedures in order to make these procedures accessible to the Presentation Manager. When preparing a dynamic-link library, the EXPORTS statement *must list the names of all functions that can be called by a client program*—that is, all public functions. Listing a function name under this statement is known as **exporting** the function, and renders the function accessible to client programs. For the example dynamic-link module, the function names all have to be given in uppercase letters since the public

functions are declared using the pascal keyword—this forces the compiler to write the function names to the object file using uppercase letters.

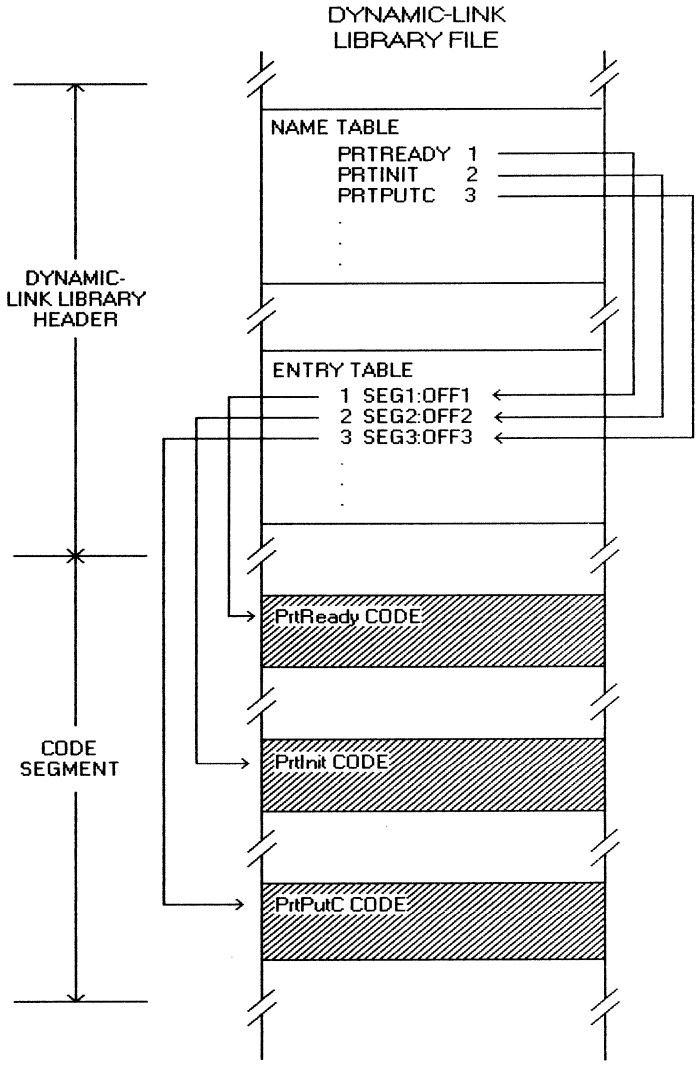
When the linker prepares the dynamic-link library, it places the names of all functions listed in the EXPORTS statement in a table of entry point names within the library file header (referred to as the **name table** in the remainder of this discussion). As you saw in Chapter 3, if a program relocation record references a dynamic-link function by entry point name, the loader must search the name table. Thus, the name of a function listed in the EXPORTS statement is the name of the function as it appears in the source code, the entry point name listed in the name table in the library file header, *and* the entry point name that is contained in the relocation record of the client program (provided that the program uses entry point names). Naming options will be explained shortly.

For each entry point name, the name table also lists the ordinal value of the function. The ordinal value is an index into the **entry table**, which is also contained in the library file header, and lists the actual segment and offset addresses of the functions within the module code. The relationship between the name table, the entry table, and the function code in the example module is illustrated in Figure 4-6. This figure depicts only the first three functions listed in the EXPORTS statement.

As you can see from Figure 4-6, if a relocation record in a program file refers to a dynamic-link function by entry point name, the system must first search the name table to obtain the function's ordinal value. It must then use the ordinal value to fetch the function address from the entry table.

Remember from Chapter 3, however, that a program relocation record can store the entry point of a dynamic-link function as an ordinal value rather than as an entry point name. In this case, the loader can obtain the function address directly from the entry table, and loading is more efficient. Also, if all calling programs use ordinal values rather than names, the name table is bypassed, and thus the system does not need to keep this table resident in memory.

Figure 4-6: Name, entry tables, and function code in an example DLL.



If you list only the name of a dynamic-link function under an EXPORTS statement, then the relocation records in all client programs subsequently prepared will reference the function by entry point name. However, you can optionally follow the function name with an ordinal value, specified as *@n*, where *n* is a unique integer value of 1 or greater. In this case, the function will be assigned the specified ordinal value, and all client programs will reference the entry point by ordinal value. Since using ordinal values is the more efficient option, all of the functions in the EXPORTS table listed above are assigned ordinal values. Note that the series of ordinal numbers you choose does not need to be contiguous—for example, you could assign the values 1, 10, 20.

You may wonder how the EXPORTS statement within the module definition file can determine the manner in which the *program* relocation records refer to the entry points of dynamic-link functions. As you will see in the next section, the IMPLIB utility uses the module definition file to generate the import library. Remember that the object records in an import library specify the entry point for each dynamic-link function either as an ordinal value or as a name. If the EXPORTS statement for a given function specifies an ordinal value, the dynamic-link record created by IMPLIB will identify the entry point by ordinal value; otherwise, it will identify it by name. When the linker resolves a reference to this function, the relocation record is assigned either an ordinal value or a name, depending upon which type of value is found in the dynamic-link record that is used to resolve the reference.

This section concludes with two additional technical notes. First, the name table referred to in this discussion actually consists of two parts: a **nonresident name table** and a **resident name table**. If a function name under the EXPORTS statement is accompanied with an ordinal value, the linker places this name in the nonresident name table (unless you include the optional keyword RESIDENTNAME after the ordinal value). The nonresident name table is not kept resident in memory since the functions it lists are normally referenced directly by ordinal value; consequently, the table is seldom used. If, however, a function name is not accompanied with an ordinal value, the linker places this name in the resident name table. This table is kept resident in memory since the

functions it lists must be referenced by name; consequently, the table is used frequently.

Second, when listing a function name under the EXPORTS statement, you can optionally specify an **internal** function name in addition to the entry name. The internal name is the function name used within the module source code, and written to the object file by the compiler. You would specify an internal name only if this name differs from the function entry point name, which is placed in the dynamic-link file name table, and may be used by client programs to refer to the function. For example, the following statement exports a function that has the internal name **privatename** and the entry point name **PublicName:**

```
EXPORTS
    PUBLICNAME=PRIVATENAME @1
```

The discussions in this section have assumed that references to dynamic-link functions are resolved through an import library. See the section on Using the Dynamic-Link Library, later in the chapter, for information on resolving these references through an IMPORTS statement in the program's module definition file.

Preparing the Dynamic-Link Library

Figure 4-4 lists a MAKE file for preparing the example dynamic-link library. An overview of the three basic steps performed by this MAKE file was given at the beginning of the chapter. This section discusses some of the details of the individual commands.

The compiler command line specifies the switch **/ASw,** which was explained in Chapter 1. As you saw, the w option tells the compiler not to assume that the SS register is equal to the DS register. For a multiple-thread application, this option is necessary because each new thread uses its own stack, which may be located in another segment. For a dynamic-link module, the w option is required even if the module does not start additional program threads. This option is needed because a dynamic-link function always uses the stack belonging to the **client program**; simply, when the function is called, the SS register contains the selector for the client's stack segment. In contrast, the DS register is loaded with the

selector of the **dynamic-link module's** data segment—due to the _loadds keyword, explained previously. Therefore, these two registers are unequal during execution of the function code.

Because the dynamic-link function borrows the stack belonging to the client program, you should also specify the /Gs option to disable stack checking within dynamic-link functions. This option is likewise required for Presentation Manager applications, and was explained in Chapter 2. Since the stack-checking routine assumes that the current stack is located within the automatic data segment, it would produce unpredictable results if called from a dynamic-link function.

Note that you can optionally use the **/ASu** option rather than /ASw. The u flag has the following two effects:

1. Like the w option, it tells the compiler not to assume the equality of DS and SS.

2. It causes the compiler to reload the DS register at the beginning of every function defined within the source file, and to restore the former DS value immediately before the function returns. This flag thus has the same effect as the _loadds keyword described earlier, except that it affects *all* functions in the file.

Since it is not necessary to reload the DS register for private functions, the example MAKE file uses the /ASw flag rather than /ASu, and then defines all public functions with the _loadds keyword.

The LINK command in the MAKE file is similar to linker commands seen in previous chapters, except that the output file name is fully specified, and contains the .DLL extension. Note, however, that you cannot generate a new dynamic-link library file while a client process is still using this library. The system does not allow you to delete, overwrite, or otherwise modify a dynamic-link library that is in use. A rationale for this restriction is that the system may need to reread read-only segments, such as code segments, that have been temporarily discarded in order to free memory. See the discussion on virtual memory in Chapter 5.

If the code for your dynamic-link library was defined in more than one C source file, you must specify all of the corresponding object files when generating the dynamic-link library with the linker.

Finally, the MAKE file uses the IMPLIB utility to generate an import library. The first parameter on the command line invoking this program is the name of the import library that is to be generated, which should have the .LIB extension. The second parameter is the name of the module definition file that is used as the source of data for generating the import library. As mentioned in the last section, the IMPLIB utility reads the EXPORTS statement within this definition file, and writes a dynamic-link record to the import library for each function listed under this statement.

As you will see in the next section, creating an import library is not mandatory since the linker can resolve references to dynamic-link functions through an IMPORTS statement within the program's module definition file. Using an import library, however, can simplify the task of preparing the application. If you are developing a commercial dynamic-link function library, including an import library with your package is an important asset.

Using the Dynamic-Link Library

Figure 4-7 lists a simple program that demonstrates the use of the functions in the example dynamic-link module. This program uses these functions to print a simulated report. First, the main function displays a menu giving the user a choice of printing the report or terminating the application.

Figure 4-7

```
/*
    Figure 4-7

    This program demonstrates the following functions defined in the example
    dynamic-link library of Figure 4-2:

        PrtReady
```

```
          PrtInit
          PrtPutS
          PrtNewPage
          PrtPosition

     The program can be built using the following commands:

          cl /c /W2 /G2 /Zp FIG4_7.C
          link /NOI /NOD FIG4_7.OBJ,, NUL, FIG4_2.LIB SLIBCE.LIB OS2.LIB;
*/
#include <STDIO.H>
#include <CONIO.H>
#include <PROCESS.H>

#include "FIG4_3.H"

void PrintReport (void);      /* Uses dynamic-link functions to print report.*/

void main (void)
     {
     int Choice;

     printf ("Programs Options:\n");              /* Display a menu.         */
     printf ("     (1)  Print Report\n");
     printf ("     (2)  Terminate Program\n");
     printf ("Select 1 or 2:  ");

     for (;;)
          switch (getch () - '0')
```

```
            {
            case 1:
                 PrintReport ();              /* Use the functions.      */
            case 2:
                 exit (0);
            }
     } /* end main */

/* Report printing data structures and functions:                         */

static void Header (void);              /* Prints report headers.         */
static int Row;                         /* NEXT row to be printed.        */

void PrintReport (void)
     {
     unsigned ErrorCode;                /* Saves error code.              */
     int i;                             /* Loop index.                    */
     unsigned char FlagReady;           /* Flag indicating printer ready. */

                                  /* Make sure that printer is ready:     */
     while (!(ErrorCode = PrtReady (&FlagReady)) && !FlagReady)
          {
          printf ("\nReady printer and press any key to continue ...");
          getch ();
          }
     if (ErrorCode)
          {
          printf ("PrtReady error %d\n",ErrorCode);
          exit (1);
```

```
     }
printf ("\nPrinting report...");

ErrorCode = PrtInit ();        /* Initialize printer.                    */

                               /* Send control code sequence for near    */
ErrorCode = PrtPutS            /* letter quality (Okidata).              */
     ("\x1b\x49\x33");         /* Control code string.                   */
if (ErrorCode)
     {
     printf ("PrtPutS error %d\n",ErrorCode);
     exit (1);
     }

ErrorCode = PrtNewPage         /* Initialize a new page without formfeed.*/
     (0);                      /* Flag indicates NO formfeed.            */
if (ErrorCode)
     {
     printf ("PrtNewPage error %d\n",ErrorCode);
     exit (1);
     }

Header ();                     /* Print first header.                    */

for (i = 1; i  80; ++i)        /* Process 80 detail lines.               */
     {
     if (Row >  55)
          {
          PrtNewPage (1);               /* New page with a formfeed.    */
          Header ();                    /* Print another header.        */
```

```
            }
        PrtPosition ("Field One",Row,1);
        PrtPosition ("Field Two",Row,23);
        PrtPosition ("Field Three",Row,44);
        PrtPosition ("Field Four",Row++,67);
        }

    PrtPosition ("End of Report",++Row,1);

    PrtNewPage (1);              /* Force out last page.                    */

    } /* end PrintReport */

static void Header (void)
    {
    PrtPosition ("S A M P L E   R E P O R T",1,27);
    PrtPosition ("Heading One",3,1);
    PrtPosition ("Heading Two",3,23);
    PrtPosition ("Heading Three",3,44);
    PrtPosition ("Heading Four",3,67);
    PrtPosition ("-----------",4,1);
    PrtPosition ("-----------",4,23);
    PrtPosition ("-------------",4,44);
    PrtPosition ("------------",4,67);
    Row = 6;
    } /* end Header */
```

If the user chooses to print the report, the function PrintReport is called, which performs the following specific tasks:

- It calls PrtReady (repeatedly if necessary) until it determines that the printer is ready to receive output.
- It calls PrtInit to reset the printer, thereby clearing any control codes previously sent.
- It calls PrtPutS to send a string of control codes that place the printer in near-letter quality mode. Note that sending these codes does not move the printer head.
- It calls PrtNewPage to initialize the module row and column counters to the start of a new page; it passes this function a value of zero so that it does not generate a formfeed.
- It now enters a loop that prints the report detail lines; each field is printed at a specific position on the page using the PrtPosition function.
- After 55 lines have been printed, it calls PrtNewPage with a non-zero parameter to reset the counters and to generate a formfeed.
- It also uses the PrtPosition function to print the header at the top of each page.

As required, the program includes the module header file listed in Figure 4-3.The program can be prepared using the following two commands at the OS/2 prompt:

```
cl /c /W2 /G2 /Zp FIG4_7.C
```

and:

```
link /NOI /NOD FIG4_7.OBJ,, NUL, FIG4_2.LIB SLIBCE.LIB OS2.LIB;
```

Note that the linker command line specifies the import library (FIG4_2.LIB, generated as described previously) to resolve references to the dynamic-link functions.

Rather than using an import library, the linker could resolve references to the dynamic-link library functions through an IMPORTS statement in a module definition file. Figure 4-8 lists a module definition file that could

be used with the example program of Figure 4-7 in place of an import library.

Figure 4-8

```
;               Figure 4-8
;               A module definition file for the program of Figure 4-7

Name FIG4_7

PROTMODE

IMPORTS
                PRTREADY        = FIG4_2.1
                PRTINIT         = FIG4_2.2
                PRTPUTS         = FIG4_2.4
                PRTPOSITION     = FIG4_2.5
                PRTNEWPAGE      = FIG4_2.6
```

If you use the module definition file to resolve references to the dynamic-link functions, you should link the program as follows:

```
link /NOI /NOD FIG4_7.OBJ,, NUL, SLIBCE.LIB OS2.LIB, FIG4_8.DEF
```

The following is an example of a line from the IMPORTS statement within the module definition file of Figure 4-8:

```
PRTINIT = FIG4_2.2
```

This line is used to import the function PrtInit. PRTINIT is the internal name of the function as it appears in the object file (remember that the compiler converts pascal function names to uppercase letters); FIG4_2 is the name of the dynamic-link library file (without extension); and the 2 following the period identifies the function entry point by its ordinal

value. When the linker writes the relocation record for this function, it identifies the entry point by ordinal value which, as you have seen, results in fast loading.

Note that the ordinal value of the function PrtInit is known because it was explicitly assigned in the EXPORTS statement of the module definition file for the dynamic-link library (Figure 4-4). All dynamic-link functions have ordinal values; however, you may not know the ordinal values of the functions in a particular dynamic-link library for one of two reasons. First, you may have exported the functions without specifying the ordinal values (if an EXPORTS statement does not specify the ordinal number of a function, the linker assigns a default value). Second, you may be calling the functions, such as an OS/2 API module, in a dynamic-link library provided by a programmer who has not published the ordinal values of the functions.

Fortunately, the ordinal values are recorded in the dynamic-link file header; you can read this header and obtain a list of all functions and their ordinal values using the Microsoft EXEHDR.EXE utility, or the program EXELOOK.EXE I have provided in the *OS/Tools* development toolkit (see the software offer in the back of the book).

If you decide not to use the ordinal value of a function you can specify the entry point by the **entry point name**. The following line demonstrates this option:

```
PRTINIT = FIG4_2.PRTINIT
```

Since the internal name (the first word in the line) is the same as the entry point name (the name following the period), you can omit the internal name, as in the following line:

```
FIG4_2.PRTINIT
```

When this example program is run, the dynamic-link library must be in a directory specified by the LIBPATH configuration command. The library is loaded, and calls to the library functions are resolved, as described in Chapter 3.

Note that this section describes **loadtime** dynamic linking. See Chapter 8 for information on **runtime** dynamic linking.

CHAPTER 5

SHARING DATA

An important feature of the dynamic-link mechanism is its support for both **instance** and **global** data segments. A dynamic-link library can have one or more instance data segments, and one or more global data segments.

Remember that a dynamic-link library can be used by several concurrent processes. The system loads a separate copy of an instance data segment for each client process. When a process calls a function in the dynamic-link library, the dynamic-link code accesses the copy of the instance data segment reserved for this process. Since each client process has a private copy of an instance data segment, such a segment is also known as a **nonshared** data segment.

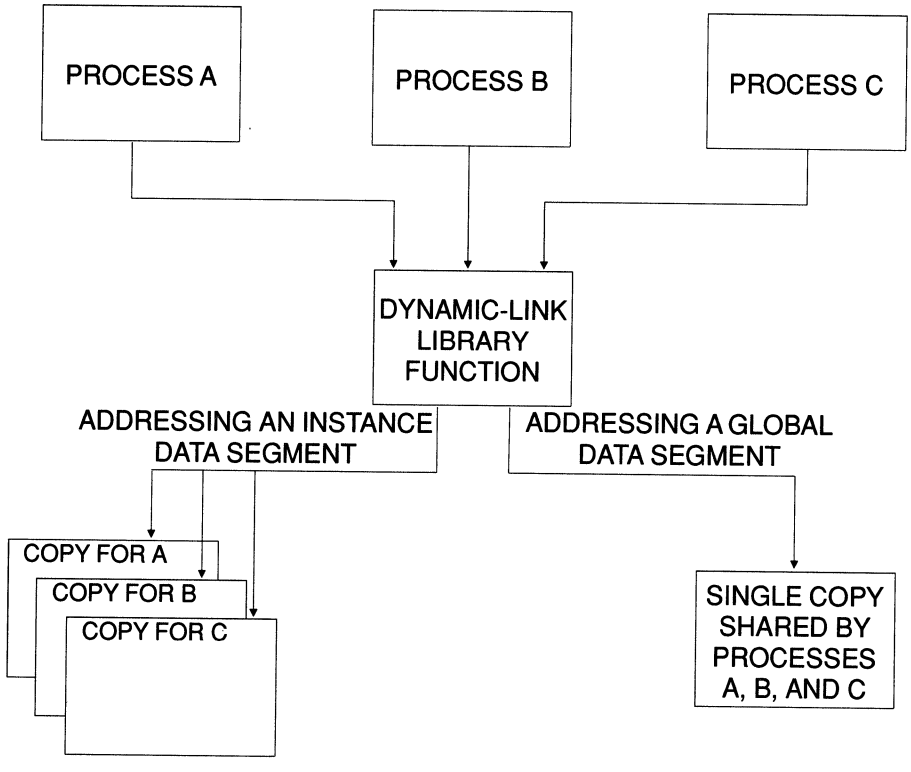
In contrast, the system loads only a single copy of a global data segment. A dynamic-link function always accesses the same copy of this segment, regardless of which process called the function. Since all processes use a common copy of a global data segment, this type of segment is also known as a **shared** data segment. Figure 5-1 illustrates these two types of segments.

The previous chapter showed how to create a dynamic-link library that used only instance data. Creating one or more global data segments, however, is an important means for sharing data among the separate

client processes and for coordinating the activities of function instances running under these separate processes.

This chapter begins by reviewing how to create dynamic-link libraries that contain only instance data segments. It then goes on to discuss how to create dynamic-link libraries that contain only global data segments. Next, it describes creating dynamic-link libraries with *both* types of data segments. Finally, it presents an example dynamic-link module that uses both instance and global data to manage a shared device—the monitor.

Figure 5-1: Instance and global data segments.



Creating an Instance Data Segment

Figure 5-2 lists the C source code for a simple dynamic-link module that serves to illustrate the essential differences between instance and global data. This module defines the external variable *Count*, initializing it to 0. Note that the term **external** means that the variable is defined outside the scope of a function. The C literature also describes such variables as **global**. This book, however, reserves the term global to describe dynamic-link library data that is located in a global data segment. The module also contains the function **PrintCount**, which first increments *Count* and prints the resulting value, and then pauses before returning control until the user presses a key.

Figure 5-2

```
/*
     Figure 5-2

     A dynamic-link module illustrating the differences between instance and

     global data items.

*/

#define INCL_SUB
#include <OS2.H>

int _acrtused = 0;              /* Define variable to avoid linking in C       */
                                /* startup code.                               */

int far Count = 0;              /* External data item placed in automatic data */
                                /* segment.                                    */

void pascal far _loadds PrintCount
     (void)
```

```
{
char Ch;                              /* Character value of 'Count'.       */
KBDKEYINFO Key;                       /* Structure to receive keyboard key.*/

Ch = (char)(++Count + '0');           /* Increment 'Count' and convert to  */
                                      /* a character.                      */

                                      /* Display message:                  */
VioWrtTTY                             /* Teletype style output.            */
     ("Count = ",                     /* String to display.                */
     8,                               /* Length of string.                 */
     0);                              /* Reserved value:  must be 0.       */

VioWrtTTY (&Ch, 1, 0);

VioWrtTTY ("\r\nPress any key to terminate program...", 39, 0);

KbdCharIn                             /* Read a character from keyboard.   */
     (&Key,                           /* Address of receiving structure.   */
     IO_WAIT,                         /* Wait until a key is pressed.      */
     0);                              /* Default keyboard handle.          */

} /* end PrintCount */
```

Figure 5-3 presents a module definition file that creates instance data when linking the dynamic-link module of Figure 5-2. Figure 5-4 provides a MAKE file for preparing the dynamic-link library as well as the associated import library.

Figure 5-3

```
;       Figure 5-3
;       A module definition file for creating instance data when
;       linking the module of Figure 5-2.

LIBRARY FIG5_2

PROTMODE

DATA MULTIPLE

EXPORTS
        PRINTCOUNT      @1
```

Figure 5-4

```
#     Figure 5-4
#     A MAKE file for preparing the dynamic-link module of Figure 5-2.
#
FIG5_2.OBJ : FIG5_2.C
     cl /c /W2 /ASw /G2s /Zp FIG5_2.C

FIG5_2.DLL : FIG5_2.OBJ FIG5_3.DEF
     link /NOI /NOD FIG5_2.OBJ, FIG5_2.DLL, FIG5_2.MAP, OS2.LIB, FIG5_3.DEF

FIG5_2.LIB : FIG5_3.DEF
     implib FIG5_2.LIB FIG5_3.DEF
```

Finally, Figure 5-5 lists a C program that calls the dynamic-link function PrintCount and terminates. You can use the following two commands to prepare this program:

```
cl /c /G2 FIG5_5.C
```

and:

```
link /NOI /NOD FIG5_5.OBJ,, NUL.LST, FIG5_2.LIB  SLIBCE.LIB OS2.LIB;
```

Figure 5-5

```
/*
     Figure 5-5

     A program that calls the dynamic-link function 'PrintCount', defined in
     Figure 5-2.
*/

void pascal far _loadds PrintCount (void);

void main (void)
     {
     PrintCount ();

     } /* end main */
```

Remember to substitute the actual name of your small-model protected-mode C library file for SLIBCE.LIB, and the name of your OS/2 API import library for OS2.LIB.

Note that the definition file of Figure 5-3 contains the statement:

```
DATA MULTIPLE
```

As explained in Chapter 4, this statement causes the automatic data segment to become an instance data segment (as you will see later in the chapter, it causes any other data segments created by the compiler to be instance segments by default). Since *Count* is contained in the automatic data segment, PrintCount accesses a *separate copy* of *Count* for each client process.

To demonstrate the properties of instance data, run the example program of Figure 5-5 from the OS/2 prompt. The program will indicate that *Count* has a value of 1 and it will pause for keyboard input. *Count* equals 1 because it was initialized to 0 and is incremented immediately before it is displayed. Before pressing a key to terminate the program, however, go into another screen group and run a second instance of the program. See the OS/2 user's reference for information on starting new screen groups, which are also known as **sessions**. The second instance will likewise display a value of 1 for the variable *Count*, indicating that a new copy of the automatic data segment was loaded when the second client process was started. If the automatic data segment were shared, *Count* would now be equal to 2.

Note that for small-data model C programs—the small and medium memory models—all external data items (such as *Count*) are normally placed in the automatic data segment. For large-data programs—that is, the compact, large, and huge memory models—only *initialized* external data and data items declared as static are placed in this segment. Uninitialized data are placed in a separate segment. Later in the chapter, you will see two ways to force the compiler to place data normally located in the automatic data segment into an alternate data segment.

Also, remember that automatic data items, those defined within the scope of a function without using the static or extern keywords, are not placed in a data segment and are never shared among separate processes. As you saw in Chapter 4, they are not shared even among separate function invocations created by the threads of a single process.

The automatic data segment consists of a group of logical segments, each one storing a specific type of data. For example, the logical segment named _DATA stores initialized external and static data, and _BSS stores uninitialized static data. For application programs, the automatic data segment also contains the program stack, which is placed in a logical

segment named STACK. Dynamic-link libraries, however, do not reserve an area for a stack since they use the stack belonging to the calling program. This group of logical segments is named DGROUP.

When the dynamic-link library file is created, the linker combines all of the logical segments contained in the automatic data segment into a single physical segment. To access data within this segment, the DS register is assigned the selector of the physical segment. Note, however, that the client program and the dynamic-link module have *separate* automatic data segments, since they were prepared separately by the linker. Therefore, when a dynamic-link function is called by a client program, it must first reload the DS register with the selector for the module's own automatic data segment, and it must restore the original DS value before the function returns control to the client. Remember that a function automatically performs the required reloading and restoring of the DS register if it is declared with the _loadds keyword, or if it is compiled with the /Au option.

Creating a Global Data Segment

Figure 5-6 lists a module definition file that contains the following DATA statement:

```
DATA SINGLE
```

This statement causes the linker to mark the automatic data segment as a global data segment (as you will see later in the chapter, it causes all other data segments defined by the compiler to become global by default). You can use this linker definition file to prepare the dynamic-link library of Figure 5-2, rather than using the definition file in Figure 5-3. The resulting dynamic-link file will have a global automatic data segment, and; therefore, the variable *Count* will be shared among all client processes.

After you have prepared the new version of the dynamic-link library, run the example client program as described in the previous section. The example program is listed in Figure 5-5. You do not have to recompile or relink this program since the dynamic-link calling interface has remained unaltered. As before, the first instance of the program will indicate that

the value of *Count* is 1. However, when you run a second instance of the program (*before* the first instance has terminated), you will see that the value of *Count* has become 2, since all client processes now share a single copy of the variable. If you start yet another instance of the program—while one or both of the previous instances are still active—you will see that *Count* becomes 3. As long as a single process using the dynamic-link module remains active, Count will continue to be incremented with each new instance. If, however, all processes that use the dynamic-link library are terminated, the library is released from memory. When a subsequent client process is run, the data segment will be reloaded from the disk and *Count* will once again be initialized to 0.

Figure 5-6

```
;       Figure 5-6
;       A module definition file for creating global data when
;       linking the module of Figure 5-2.

LIBRARY FIG5_2

PROTMODE

DATA SINGLE

EXPORTS
        PRINTCOUNT     @1
```

At this point, you may wonder how OS/2 manages global and instance data segments. Remember that a dynamic-link code segment is always shared among all client processes; like most code segments, it generally contains a set of hard-coded addresses of items in the data segment. To answer the question how does a single data address access one segment

when used by a given process, and another segment when used by a different process, we must digress into the topic of virtual memory.

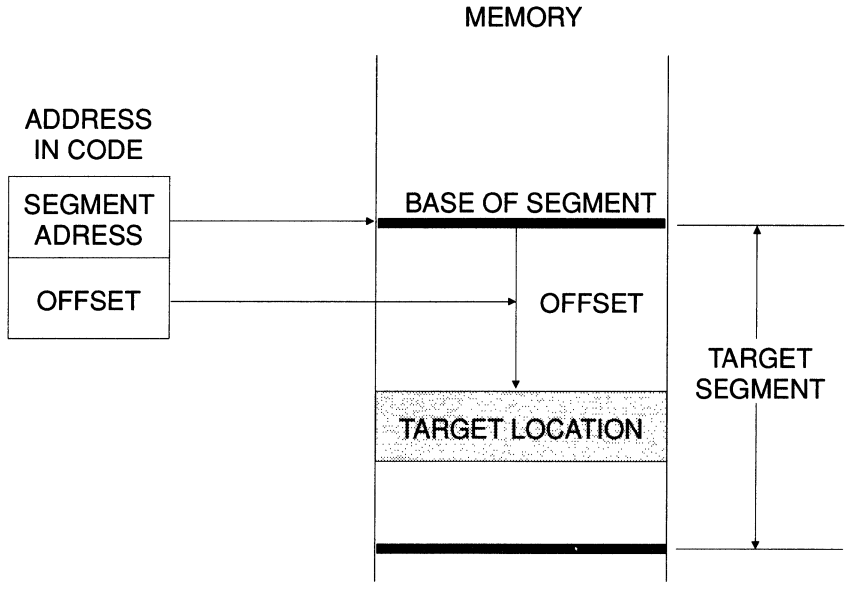
Virtual Memory

The term **virtual memory** refers to the logical address space that OS/2 maintains and makes available to protected-mode application programs. The virtual memory mechanism allows OS/2 to provide an address space that is larger than the amount of physical memory installed in the machine. When physical memory is exhausted, the system automatically swaps segments to the hard disk. It also permits the operating system to maintain important information on each memory segment, which is used to protect the integrity of the system.

To an application program, the virtual memory mechanism is largely invisible; the program simply accesses the memory it has been allocated, and the operating system automatically performs all the details of maintaining segment information and swapping segments to the disk. The program, however, must follow certain rules; for example, it cannot write to a code segment. In the same manner as a real-mode program, a memory location is addressed using a 16-bit segment value in combination with a 16-bit offset. Under the real-mode addressing method, the segment value is the actual physical address of the base of the segment in memory. More accurately, the physical address is obtained by multiplying the segment value by 16.

In the virtual memory mechanism, however, the segment value is known as a segment **selector**, and is an index into a table containing the actual physical segment address as well other information on the segment. This table is known as a **descriptor table**, and the individual entries for each segment are known as **segment descriptors**. Figure 5-7 illustrates the direct real-mode addressing scheme, and Figure 5-8 shows the indirect addressing method used to support virtual memory.

Figure 5-7: The real-mode addressing method.

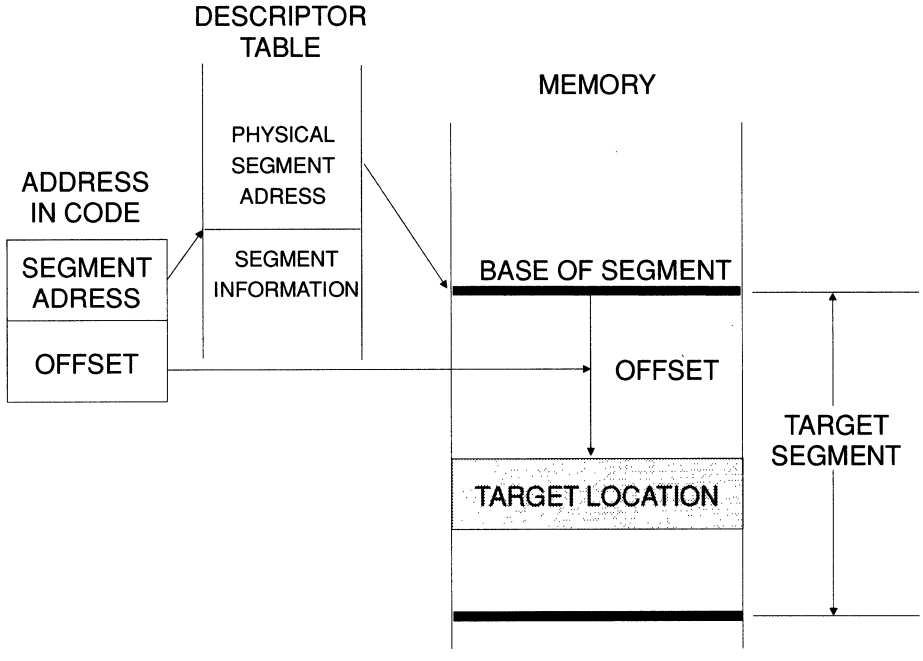


The following are among the more important items of segment information stored within a segment descriptor:

- A flag indicating whether the segment is a code segment or a data segment. The operating system will not allow an application to write to a segment marked as a code segment; thus, separate processes can safely share dynamic-link code segments without danger of an aberrant process corrupting the common code.
- A flag indicating whether the segment is present in memory or has been swapped to the disk. This information allows the system to automatically fetch required segments.

- The length of the segment. This field allows the system to prevent programs from addressing beyond the end of an allocated segment.

Figure 5-8: The Virtual Memory addressing method.



The operating system maintains two types of descriptor tables. First, each process owns a private **local descriptor table**, which can be accessed only by the operating system and by the process that owns it. Second, the system maintains a single **global descriptor table**, which can be accessed by any process. OS/2 places the descriptors for all application segments, even those which refer to shared memory segments within the local descriptor table. The term **application segments** refers to the code and data segments defined within an executable program or

dynamic-link file, as well as segments dynamically allocated from the operating system at runtime. The global descriptor table is used for special-purpose segments, such as the **global information segment**, which contains a variety of information pertaining to the entire system. A program can obtain the selector of this segment by calling the API function **DosGetInfoSeg**; this selector is an index into the global descriptor table.

To grant a process access to an application segment, the system performs the following two actions:

1. It sets up a descriptor for the segment in the local descriptor table belonging to the process.

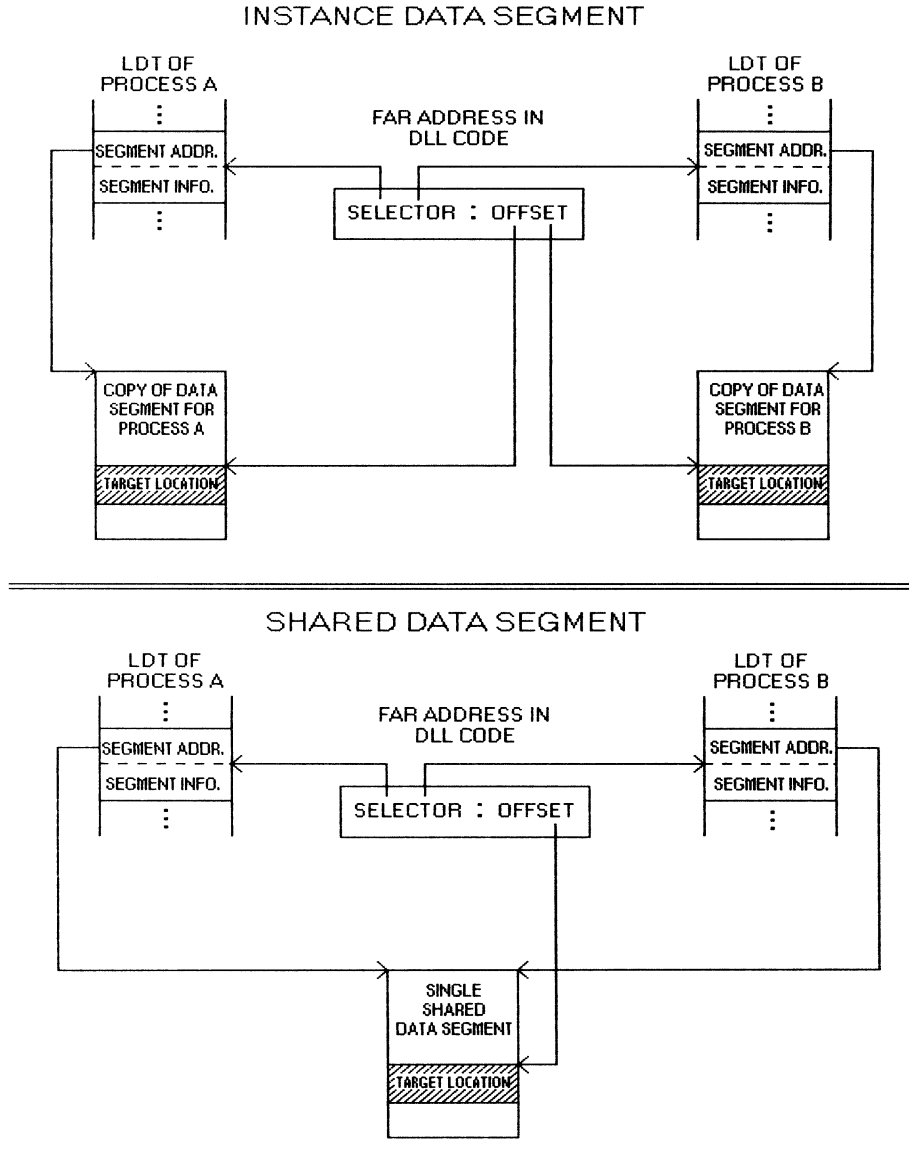
2. It provides the process a selector for this segment, which is an index that points to the newly created segment descriptor in the local descriptor table.

When the system runs the *first* process referring to a given dynamic-link library, it loads all of the dynamic-link module's code and data segments into memory and grants the process access to these segments using the above two steps. When the system runs a *subsequent* client process, it loads new copies of all instance data segments and provides access to these segments. For all code and global data segments, however, it does not load new copies, rather it grants the process access to the segments already in memory.

Consider an address, of a code or data location within the code belonging to a dynamic-link function that is called by two client processes. When executing this function, both processes see the *same* address value; each process, however, uses the selector portion of the address as an index into its *own* local descriptor table. If the target segment is an instance data segment, each descriptor will point to (that is, contain the physical address of) a distinct segment in memory. If, however, the target segment is a code or global data segment, both descriptors will point to the *same* segment in memory. This arrangement is illustrated in Figure 5-9.

For more information on how OS/2 assigns selectors for segments defined by dynamic-link modules, see the discussion on the Disjoint Descriptor Space in Chapter 8.

Figure 5-9: Descriptors for nonshared and shared segments.



Creating Instance and Global Data Segments

The previous sections in this chapter have shown how to define the instance attribute of the automatic data segment so that a program will have only instance data or only global data. Typically an application that uses a global data segment requires one or more instance segments for storing information private to each client process. Fortunately, a single dynamic-link library can have both instance and global data segments.

To create a dynamic-link module with both types of data segments, you need to perform the following two basic steps:

1. You must create one or more segments in addition to the automatic data segment, and make sure that specific data items are placed in the correct segment.

2. You must specify for each segment whether it is an instance or a global data segment.

The discussions in this chapter assume that instance data items are more commonly required than global data items. Accordingly, the techniques presented in this section are for creating a dynamic-link module with an instance automatic data segment, and one or more additional segments that are global. Thus, most data items will automatically become instance data; to create a global data item, you must explicitly assign it to a global segment.

Creating additional data segments and assigning instance attributes to these segments is more difficult in C than in assembly language. This section, however, presents two strategies that can be used with Microsoft C. The first strategy employs a separate program source file to define shared data, and the second strategy defines both types of data within a single C source file.

Using Two Source Files

Figures 5-10 through 5-13 provide a set of files for creating a simple dynamic-link library containing both global and instance data. The dynamic-link module is defined in two C source files, which are listed in

Figures 5-10 and 5-11. Figures 5-12 and 5-13 provide a module definition file and a MAKE file for preparing the example module.

The source file in Figure 5-10 defines the variable *PrivateCount*, initializing it to 0. Like the variable *Count* in the program of Figure 5-2, *PrivateCount* is placed in the automatic data segment. This source file also contains the function PrintCount, which increments and prints the resulting values of the variables *PrivateCount* and *PublicCount*.

Figure 5-10

```
/*
     Figure 5-10

     The primary source file for creating a dynamic-link library that

     demonstrates both instance and global data.
*/

#define INCL_SUB
#include <OS2.H>
int _acrtused = 0;              /* Define variable to avoid linking in C      */
                                /* startup code.                              */

extern int far PublicCount;     /* Global data item in data segment JEROME;   */
                                /* this variable is defined in the secondary  */
                                /* source file of Figure 5-11.                */

int PrivateCount = 0;           /* Instance data item in automatic data segment*/

void pascal far _loadds PrintCount
     (void)
     {
```

```
    char Ch;                        /* Holds character value of counters.     */
    KBDKEYINFO Key;                 /* Receives keyboard information.         */

    /*** Print value of instance data item 'PrivateCount'. ******************/
    VioWrtTTY ("PrivateCount = ", 15, 0);
    Ch = (char)(++PrivateCount + '0');
    VioWrtTTY (&Ch, 1, 0);

    /*** Print value of global data item 'PublicCount'. *********************/
    VioWrtTTY ("\r\nPublicCount = ", 16, 0);
    Ch = (char)(++PublicCount + '0');
    VioWrtTTY (&Ch, 1, 0);

    /*** Pause before returning control. ************************************/
    VioWrtTTY ("\r\nPress any key to terminate program...", 39, 0);

    KbdCharIn (&Key, IO_WAIT, 0);

    } /* end PrintCount */
```

Figure 5-11

```
/*
    Figure 5-11

    A secondary source file for defining global data for the dynamic-link
    library defined in Figure 5-10.
*/

int PublicCount = 0;  /* Data item placed in the global data segment JEROME. */
```

Figure 5-12

```
;       Figure 5-12
;       Module definition file for linking the module of Figures
;       5-10 and 5-11.
;
LIBRARY FIG5_10

PROTMODE

DATA MULTIPLE

SEGMENTS
        JEROME          CLASS 'FAR_DATA' SHARED
        JEROME_CONST    CLASS 'FAR_DATA' SHARED
        JEROME_BSS              CLASS 'FAR_DATA' SHARED

EXPORTS
        PRINTCOUNT              @1
```

Figure 5-13

```
#     Figure 5-13
#     A MAKE file for preparing the dynamic-link module defined in Figures
#     5-10 and 5-11.
   #
FIG5_10.OBJ : FIG5_10.C
     cl /c /W2 /ASw /G2s /Zp FIG5_10.C

FIG5_11.OBJ : FIG5_11.C
     cl /c /W2 /ASw /G2s /Zp /ND JEROME FIG5_11.C


FIG5_10.DLL : FIG5_10.OBJ FIG5_11.OBJ FIG5_12.DEF
```

```
    link /NOI /NOD FIG5_10 FIG5_11, FIG5_10.DLL, FIG5_10, OS2, FIG5_12

FIG5_10.LIB : FIG5_12.DEF
    implib FIG5_10.LIB FIG5_12.DEF
```

The variable *PublicCount* is defined (equal to 0) in the secondary source file listed in Figure 5-11 so that it can be placed in a segment other than the automatic data segment. Normally, the data defined in this file would be located within the same automatic data segment as the data defined in the main source file. However, the MAKE file (Figure 5-13) compiles the secondary source file using the **/ND** option, as follows:

```
cl /c /W2 /ASw /G2s /Zp /ND JEROME FIG5_11.c
```

The /ND flag in this command causes the compiler to place all data items normally included in the automatic data segment into a separate segment named JEROME. Accordingly, the external variable *PublicCount* is placed in the data segment JEROME, rather than in the automatic data segment. Since uninitialized external data in a large-data model program is not placed in the automatic data segment, it will not be placed in the segment JEROME, even if defined within the secondary source file.

The only tasks remaining are to make the automatic data segment an instance segment, and to make the segment named JEROME a global segment. This is done so that *PrivateCount* becomes an instance data item, and *PublicCount* becomes a global data item. These tasks are accomplished in the module definition file of Figure 5-12.

First, the following statement:

```
DATA MULTIPLE
```

makes the automatic data segment an instance segment. As you will see in the following section, this statement also makes all other segments instance segments by default. This default attribute; however, can be

overridden for a specific segment by including a SEGMENTS command. The example program overrides the default condition and makes JEROME a global segment through the following statement:

```
SEGMENTS
  JEROME              CLASS 'FAR_DATA' SHARED
  JEROME_CONST        CLASS 'FAR_DATA' SHARED
  JEROME_BSS          CLASS 'FAR_DATA' SHARED
```

The SHARED keyword renders the specified segment a global segment (the NONSHARED keyword would make it an instance segment). When you specify a segment with the /ND flag, the compiler creates two additional logical segments, which are given the suffixes _CONST and _BSS. Since all three logical segments are placed in the same segment group, and are therefore linked into a single physical segment, you must include all three segments in the SEGMENTS command. You must specify the segment class names (the class determines how segments are ordered in memory). You can obtain the exact names of all segments and classes that the compiler creates by consulting the map file generated by the linker. When linking the dynamic-link library, simply specify a filename other than NUL for the list file, which is the third main parameter on the LINK command line. In the resulting dynamic-link library, the variable *PrivateCount* is located in an instance data segment, and the variable *PublicCount* is located in a global segment. *PublicCount* is declared in the main program as follows:

```
extern int far PublicCount;
```

The extern keyword indicates that the variable is defined within a separate source file, and the far keyword is necessary because the variable is located in a segment other than the automatic data segment and therefore must be accessed using a far address. In contrast, data in the automatic data segment can be referenced through near addresses since the DS register already contains the selector of this segment. Note that this segment arrangement enhances the efficiency of accessing instance data, generally the more common type of data. To test this dynamic-link library, you can compile and link the program listed in Figure 5-5 using the appropriate import library, as in the following commands:

```
cl /c /G2 FIG5_5.C
```

and:

```
link /NOI /NOD FIG5_5.OBJ,, NUL.LST, FIG5_10.LIB SLIBCE.LIB OS2.LIB;
```

When you call the function PrintCount from separate concurrent processes, as described previously in the chapter, *PrivateCount* is reinitialized with each new client, whereas all processes access a single shared copy of *PublicCount*.

Using a Single Source File

The listing in Figure 5-14 is the same as the listing in Figure 5-10 (described in the previous section), except that it defines both the instance data item *PrivateCount* and the global data item *PublicCount* in the same *C source* file. Accordingly, the program does not use an auxiliary source file for defining global data. Figure 5-15 contains the module definition file, and Figure 5-16 provides a MAKE script for preparing the dynamic-link library.

The variable *PublicCount* is defined within the single source file as follows:

```
int far PublicCount = 0;
```

Figure 5-14

```
/*
    Figure 5-14

    A dynamic-link module source file that defines both instance and global
    data.
*/
```

```
#define INCL_SUB
#include <OS/2.H>
int _acrtused = 0;              /* Define variable to avoid linking in C       */
                                /* startup code.                               */

int far PublicCount = 0;        /* Global data item in a separate segment.     */

int PrivateCount = 0;           /* Instance data item in automatic data segment*/

void pascal far _loadds PrintCount
     (void)
     {
     char Ch;                        /* Holds character value of counters.     */
     KBDKEYINFO Key;                 /* Receives keyboard information.         */

     /*** Print value of instance data item 'PrivateCount'. ******************/
     VioWrtTTY ("PrivateCount = ", 15, 0);
     Ch = (char)(++PrivateCount + '0');
     VioWrtTTY (&Ch, 1, 0);

     /*** Print value of global data item 'PublicCount'. *********************/
     VioWrtTTY ("\r\nPublicCount = ", 16, 0);
     Ch = (char)(++PublicCount + '0');
     VioWrtTTY (&Ch, 1, 0);

     /*** Pause before returning control. *************************************/
     VioWrtTTY ("\r\nPress any key to terminate program...", 39, 0);

     KbdCharIn (&Key, IO_WAIT, 0);

     } /* end PrintCount */
```

Figure 5-15

```
;               Figure 5-15
;               Module definition file for linking the module of Figure 5-14
;
LIBRARY FIG5_14


PROTMODE


DATA MULTIPLE SHARED


EXPORTS
               PRINTCOUNT     @1
```

Figure 5-16

```
#     Figure 5-16
#     A MAKE script for preparing the dynamic-link module of Figure 5-14
#
FIG5_14.OBJ : FIG5_14.C
     cl /c /W2 /ASw /G2s /Zp FIG5_14.C


FIG5_14.DLL : FIG5_14.OBJ FIG5_15.DEF
     link /NOI /NOD FIG5_14.OBJ, FIG5_14.DLL, NUL.MAP, OS2.LIB, FIG5_15.DEF


FIG5_14.LIB : FIG5_15.DEF
     implib FIG5_14.LIB FIG5_15.DEF
```

Using the far keyword forces the compiler to place this variable in a segment other than the automatic data segment, such placement will occur in any memory model.

The name of the segment containing PublicCount is unimportant, because you can make the automatic data segment an instance segment and make all other segments global, regardless of their names. The assignment of segment attributes is achieved through the following DATA statement in the module definition file of Figure 5-15:

```
DATA MULTIPLE SHARED
```

This statement causes the linker to make the automatic segment an instance segment, and to make all other segments global by default. The DATA statement specifies values for two fields: the **instance** field and the **shared** field. The instance field consists of either the SINGLE or the MULTIPLE keyword and, in general, determines whether the automatic data segment is global or instance. You can also specify NONE if there is no automatic data segment. The shared field consists of the SHARED or the NONSHARED keyword, and serves in general to specify whether all segments other than the default data segment are to be global or instance. Both of these fields are optional, and the values you specify (or omit) interact in complex and mysterious ways. Table 5-1 shows the effect of each possible combination.

Table 5-1: Instance and Shared Fields Effects

INSTANCE FIELD	SHARED FIELD	AUTOMATIC DATA SEGMENT[1]	OTHER SEGMENTS[2]
<none>[3]	<none>[4]	global	global
<none>	SHARED	global	global
<none>	NONSHARED	instance	instance
SINGLE	<none>	global	global
SINGLE	SHARED	global	global
SINGLE	NONSHARED	global	instance

Table 5-1: Instance and Shared Fields Effects

INSTANCE FIELD	SHARED FIELD	AUTOMATIC DATA SEGMENT[1]	OTHER SEGMENTS[2]
MULTIPLE	<none>	instance	instance
MULTIPLE	SHARED	instance	global
MULTIPLE	NONSHARED	instance	instance

1. Cannot be overridden with a SEGMENTS statement.
2. Can be overridden with a SEGMENTS statement.
3. Neither SINGLE nor MULTIPLE is specified.
4. Neither SHARED nor NONSHARED is specified.

The global/instance status specified by the DATA statement for any segment other than the automatic data segment can be overridden using a SEGMENTS statement as described in the previous section. You cannot, however, override the status specified for the automatic data segment. If you give a contradictory value in a SEGMENTS statement for any logical segment in the automatic data segment group, the compiler issues a warning and ignores the attempted override.

As in the example given in the previous section, *PrivateCount* is placed in the automatic data segment and is therefore an instance data item. *PublicCount*, however, has been forced into a segment other than the automatic data segment and is therefore a global data item.

You can test this version of the example dynamic-link library by compiling and linking the program of Figure 5-5 with the current version of the import library, as follows:

```
cl /c /G2 FIG5_5.C
```

and:

```
link /NOI /NOD FIG5_5.OBJ,, NUL.LST, FIG5_14.LIB SLIBCE.LIB OS2.LIB;
```

When you run the resulting executable program, you will discover that the dynamic-link library behaves in the same manner as the example module given in the previous section.

Defining both global and instance data within a single source file seems to be an equivalent, but easier, method than using separate source files as described in the previous section. There is a difference in these two methods, however, that may be important for certain dynamic-link libraries. The two-file method makes all data segments instance segments except for the segment specifically mentioned in the SEGMENTS statement. Accordingly, uninitialized external data in large-data model programs would automatically become instance data. Such data are placed neither in the automatic data segment nor in the segment specified by the /ND command.

In contrast, the single-file method makes all data segments global except for the automatic data segment. Accordingly, uninitialized external data in large-model programs would become global.

You can choose the method that suits the needs of your dynamic-link library. Also, the methods explored in this chapter are only two of many possible ways of combining instance and global data within a dynamic-link library.

Using Instance and Global Data Segments

This final section presents a simple dynamic-link library that uses both instance and global data. The module uses instance data to store information specific to each client process, and it uses global data to control access to a shared device—the monitor. Although the module performs only a trivial task, it serves to illustrate the basic manner in which these two types of data are used by an OS/2 **subsystem**, which is an integrated set of functions that manage a device shared by multiple processes, such as the Kbd, Mou, and Vio API services).

Figure 5-17 lists the source code for the example dynamic-link library, and Figures 5-18 and 5-19 provide a module definition file and a MAKE script respectively.

Figure 5-17

```
/*
     Figure 5-17
     An example dynamic-link module that uses both instance and global data,
     and synchronizes messages written to the screen.
*/

#define INCL_BASE
#include <OS2.H>

/*** Global Data ************************************************************/
ULONG far Sem = 0;              /* Semaphore:  a global data item placed in a  */
                                /* segment OTHER than the automatic data       */
                                /* segment.

/*** Instance Data **********************************************************/
PIDINFO ProcessID =             /* Structure for receiving process ID:  an     */
     {0, 0, 0};                 /* instance data item in the auomatic data     */
                                /* segment.                                    */

                                /* Message to print to screen:  an instance    */
                                /* data item in the automatic data segment:    */
char ProcessMessage [] = "*          Hello from Process                 *\r\n";

int _acrtused = 0;              /* Define variable to avoid linking in C       */
                                /* startup code.                               */

/*** Private functions ******************************************************/
```

```
void PrintMessage (void);      /* Prints a message on the screen.              */
void UShortToString (USHORT Source, char *Target);  /* Converts an unsigned  */
                                                    /* integer to a string.  */

void pascal far _loadds Message
     (void)
     {
     /*** Obtain current process ID when a process first calls this function.*/
     if (ProcessID.pid == 0)
          {
          DosGetPID (&ProcessID);        /* Provides the process ID.          */
                                         /* Convert numeric ID to a string:   */
          UShortToString (ProcessID.pid, ProcessMessage+30);
          }

     DosSemRequest              /* Wait for semaphore to clear and then set it.*/
          (&Sem,                /* Semaphore handle (its far address).         */
          -1L);                 /* Wait flag:  -1 means wait forever.          */

     DosSleep (500L);           /* Create a 1/2 second pause.                  */

     PrintMessage ();           /* Print a message box on the screen.          */

     DosSemClear                /* Clear the semaphore.                        */
          (&Sem);               /* Handle to global semaphore 'Sem'.           */

     DosSleep (0L);             /* Yield remainder of time slice.              */

     } /* end Message */
```

```
void PrintMessage (void)
     {
     VioWrtTTY("*********************************************\r\n",49,0);
     VioWrtTTY("*                                           *\r\n",49,0);
     VioWrtTTY("*                                           *\r\n",49,0);
     VioWrtTTY("*                                           *\r\n",49,0);
     VioWrtTTY("*                                           *\r\n",49,0);
     VioWrtTTY (ProcessMessage,49,0);
     VioWrtTTY("*                                           *\r\n",49,0);
     VioWrtTTY("*                                           *\r\n",49,0);
     VioWrtTTY("*                                           *\r\n",49,0);
     VioWrtTTY("*        press Ctrl-C to terminate          *\r\n",49,0);
     VioWrtTTY("*********************************************\r\n",49,0);

     } /* end PrintMessage */

void UShortToString (USHORT Source, char *Target)
/*
     This function converts the USHORT value 'Source' to a string, which is
     written to the buffer 'Target'.  The resulting string is NOT NULL
     terminated.
*/
     {
     register int i = 0;
     char Temp [6];

     do
          {
          Temp [i++] = (char)(Source % 10 + '0');
          Source /= 10;
```

```
        }
    while (Source);

    do
        {
        *Target++ = Temp [--i];
        }
    while (i);

    } /* end UShortToString */
```

Figure 5-18

```
;               Figure 5-18
;               A module definition file for linking the dynamic-link
;               module of Figure 5-17.
;
LIBRARY FIG5_17

PROTMODE

DATA MULTIPLE SHARED

EXPORTS
                MESSAGE @1
```

The source file of Figure 5-17 defines the function **Message**, which creates a pause and prints a message box on the screen in the same manner as the example programs given in Chapter 1. The example dynamic-link library defines both instance and global data items within the single source file using the second of the two methods described in the previous section.

Figure 5-19

```
#     Figure 5-19
#     A MAKE file for preparing the dynamic-link module of Figure 5-17.
#
FIG5_17.OBJ : FIG5_17.C
     cl /c /W2 /ASw /G2s /Zp FIG5_17.C

FIG5_17.DLL : FIG5_17.OBJ FIG5_18.DEF
     link /NOI /NOD FIG5_17.OBJ, FIG5_17.DLL, NUL.MAP, OS2.LIB, FIG5_18.DEF

FIG5_17.LIB : FIG5_18.DEF
     implib FIG5_17.LIB FIG5_18.DEF
```

The source file defines two instance data items: the PIDINFO structure **ProcessID** and the string **ProcessMessage**. Both of these items are defined as simple external data so that they are placed within the automatic data segment, which is made an instance segment through the DATA command in the linker definition file.

The ProcessID structure (specifically, the pid field) is used to receive the identification number of the current process from the API function **DosGetPID**, the first time the function Message is called by a given process. If the pid field of the ProcessID structure equals 0—its initial value—the function knows that DosGetPID has not yet been called to obtain the ID of the current process. Message then converts the numeric process ID value to an ASCII string, which is written to the string ProcessMessage. The conversion is performed by the private function **UShortToString**, and the resulting message string, ProcessMessage, is printed as one of the lines in the message box. The data items ProcessID

and ProcessMessage are both made instance items because they are used to store distinct values for each client process.

The dynamic-link module source file also defines the following global data item:

```
ULONG far Sem;
```

As discussed in the previous section, the keyword far forces the compiler to place this item in a segment other than the automatic data segment. Since the DATA command in the module definition file makes all such segments global, Sem becomes a global data item. Sem is used as a semaphore to prevent more than one process (or more than one thread within a single process) from attempting to print a message box on the screen at a given time. The methods for using a semaphore for this purpose are discussed in Chapter 1, in the section on Adding Interprocess Communication. The salient feature then is that Sem must be defined as a global data item so that any process in the system can test and set its value.

To test whether the dynamic-link function Message successfully prevents more than one process from attempting to print a message box on the screen at a given time, you must run two or more processes that call this function within the same screen group, since all processes in a given screen group share the same physical screen. OS/2 automatically segregates screen output from processes within separate screen groups.

You can test the dynamic-link module by running the programs listed in Figures 5-20 and 5-21 simultaneously within the same screen group. Both of these programs repeatedly call the dynamic-link function Message to print a series of message boxes containing the ID of the calling process. Before calling Message, however, the program of Figure 5-20 calls **DosExecPgm** to execute the program of Figure 5-21 as a child process within the same screen group. Therefore, to run both programs within a given screen group, you need only execute the program of Figure 5-20 from the OS/2 command prompt.

Figure 5-20

```
/*
     Figure 5-20
     A program that starts the program of Figure 5-21 as a child process, and
     tests the dynamic-link function 'Message'.
*/

#define INCL_BASE
#include <OS2.H>

void pascal far _loadds Message (void); /* Dynamic-link function defined in  */
                                        /* Figure 5-17.                     */

void main (void)
     {
     CHAR FailName [13];                /* Used by 'DosExecPgm'.            */
     RESULTCODES Results;               /* Used by 'DosExecPgm'.            */

     DosExecPgm                     /* Run a program as a child process.    */
          (FailName,                /* Receives name of file causing failure. */
          sizeof (FailName),        /* Length of 'FailName'.                */
          EXEC_ASYNC,               /* Execute child asynchronously.        */
          0,                        /* No arguments.                        */
          0,                        /* Environment:  0 means use parent's.  */
          &Results,                 /* Receives process ID of child.        */
          "FIG5_21.EXE");           /* Executable file name of child process. */

     for (;;)
          Message ();               /* Call dynamic-link function.          */

     } /* end main */
```

Figure 5-21

```
/*
     Figure  5-21
     A program that runs as a child process of the program defined in
     Figure 5-20, and tests the dynamic-link function 'Message'.
*/

void pascal far _loadds Message (void); /* Dynamic-link function defined in  */
                                        /* Figure 5-17.                     */
void main (void)
     {
     for (;;)
          Message ();                   /* Call dynamic-link function.      */

     } /* end main */
```

You should compile and link each of these programs separately, using the appropriate import library. For example, the following two commands can be used to prepare the program of Figure 5-20:

```
cl /c /G2 FIG5_20.C
```

and:

```
link /NOI /NOD FIG5_20.OBJ,, NUL.LST, FIG5_17.LIB SLIBCE.LIB OS2.LIB;
```

When you run the program of Figure 5-21, you will see distinct message boxes appearing alternately from each of the two processes.

CHAPTER 6

INITIALIZATION AND TERMINATION

This chapter introduces two additional options available to dynamic-link libraries: initialization routines and termination routines. In general, initialization and termination routines allow a dynamic-link module to keep track of multiple client processes. They inform the module whenever a new client process is started and whenever an existing client terminates. Accordingly, they are especially useful for dynamic-link subsystems—collections of routines that manage multiple processes sharing a common device or other system resource.

A dynamic-link module can provide a normal dynamic-link function that a client program calls before using the facilities of the module, and another function that it calls when it has completed using the module. For example, the Presentation Manager provides the functions **WinInitialize** and **WinTerminate**, explained in Chapter 2. The initialization and termination routines discussed in this chapter, however, are quite different. These functions are not explicitly called by the client program; rather, once they have been properly defined and installed, they are called automatically by the system. An initialization routine is automatically called before the client program even begins running, and a termination routine is automatically called when the client terminates, normally or abnormally.

Initialization and termination routines thus allow a dynamic-link module to smoothly manage the arrivals and departures of multiple client processes, without depending upon these clients to explicitly call entry and exit routines. Initialization and termination routines enhance the ability of a dynamic-link library to maintain its integrity, and to continue providing services to its client processes despite the misbehavior of an individual client. This arrangement is in keeping with the general philosophy of OS/2 that a single miscreant process should never be able to sabotage its fellow processes.

Writing Initialization Routines

Figures 6-1 through 6-4 provide a set of files for creating a dynamic-link module with an initialization routine that is called each time the module is referenced by a new client program. These files illustrate the following three basic steps for setting up an initialization routine:

1. Define the initialization entry point in an assembly language file.
2. Write the body of the initialization routine in the C source file.
3. Specify when the initialization routine is to be called in the module definition file.

Define the Entry Point

Figure 6-1 lists an assembly language file that defines the initialization entry point. Unlike a C source file, an assembly language source file can specify an entry point through the END statement. All assembly language files must be terminated with an END statement. This statement can consist of the END keyword alone, *or* the END keyword followed by the address of the entry point, which can be given as a code label or as the name of a procedure. The specified entry point is ultimately written to the executable file header.

For an application program, the entry point is the address that is first given control when the program is run. For C application programs, an END statement specifying the program entry point is contained in the C startup source file—CRT0.ASM.

For a dynamic-link library, the entry point is the starting address of the initialization routine. When creating a dynamic-link module, specifying an entry point is optional; if you do not specify one, the resulting module will simply not have an initialization routine. Also, if more than one assembly language file is linked together, only one of them may specify an entry point.

Figure 6-1

```
;     Figure 6-1
;     This file defines the initialization entry point for the dynamic-link
;     module of Figures 6-1 and 6-2
;
.MODEL LARGE
.CODE

EXTRN _InitRoutine : FAR            ;Initialization routine defined in the C
                                    ;source file of Figure 6-2

ENTRYPOINT PROC FAR

     call _InitRoutine              ;Call main initialization routine.
     ret                            ;FAR return to the system.

ENTRYPOINT ENDP

END ENTRYPOINT                      ;Defines the initialization entry point.
```

Figure 6-2

```
/*
     Figure 6-2
     This C source file defines the initialization routine 'InitRoutine' and
     the dynamic-link function 'DynaFunc'.
*/

#define INCL_SUB
#include <OS2.H>

/*** Global data: ************************************************************/

USHORT far ProcessCount = 0;  /* Maintains a count of new client processes.  */

/*** Instance data: **********************************************************/

char *Message;                /* Pointer to message displayed by
                              /* 'InitRoutine'.                              */

int _acrtused = 0;            /* Define variable to avoid linking in C       */
                              /* startup code.                               */

/*** Private function: *******************************************************/

void UShortToString (USHORT Source, char *Target);  /* Converts an unsigned  */
                                                    /* integer to a string.  */

/*** Initialization routine: *************************************************/
```

```
int far InitRoutine
    (void)
    {
    ++ProcessCount;                 /* Increment count of new client          */
                                    /* processes.                             */

    if (ProcessCount == 1)          /* First client process to activate the   */
                                    /* initialization routine.                */
        {
        Message =| "    First call to initialization routine.      |\r\n";
        }
    else                            /* Not the first client process.          */
        {
        Message =| " Call to initialization routine number:       |\r\n";
        UShortToString (ProcessCount, Message + 42);
        }

    VioWrtTTY ("                                                   \r\n", 50, 0);
    VioWrtTTY ("                                                   \r\n", 50, 0);
    VioWrtTTY ("                                                   \r\n", 50, 0);
    VioWrtTTY (Message, 50, 0);
    VioWrtTTY ("                                                   \r\n", 50, 0);
    VioWrtTTY ("                                                   \r\n", 50, 0);
    VioWrtTTY ("                                                   \r\n", 50, 0);
    VioWrtTTY ("\r\n", 50, 0);

    return (1);                     /* Return value of 1 indicates that the   */
                                    /* initialization routine was successful. */
```

```
    } /* end InitRoutine */

/*** Dynamic-link routine: ************************************************/

void pascal far _loadds DynaFunc (void)
    {
    KBDKEYINFO Key;                    /* Structure to receive keyboard key.*/

    VioWrtTTY ("\r\n", 50, 0);
    VioWrtTTY ("                                                  \r\n", 50, 0);
    VioWrtTTY ("                                                  \r\n", 50, 0);
    VioWrtTTY ("       Dynamic-link function called.              \r\n", 50, 0);
    VioWrtTTY ("                                                  \r\n", 50, 0);
    VioWrtTTY ("                                                  \r\n", 50, 0);
    VioWrtTTY ("         press any key to continue                \r\n", 50, 0);
    VioWrtTTY ("\r\n", 50, 0);

    KbdCharIn (&Key, IO_WAIT, 0);      /* Pause for keyboard input.         */

    } /* end DynaFunc */

/*** Private function: ****************************************************/

void UShortToString (USHORT Source, char *Target)
/*
    This function converts the USHORT value 'Source' to a string, which is
    written to the buffer 'Target'.  The resulting string is NOT NULL
    terminated.
*/
    {
```

```
register int i = 0;
char Temp [6];

do
     {
     Temp [i++] = (char)(Source % 10 + '0');
     Source /= 10;
     }
while (Source);

do
     {
     *Target++ = Temp [--i];
     }
while (i);

} /* end UShortToString */
```

Figure 6-1 contains the END statement:

```
END ENTRYPOINT
```

where ENTRYPOINT is the name of the procedure that the system is to invoke as the initialization routine. This procedure calls the function **InitRoutine**, which is defined in the C source code of Figure 6-2 and contains the main body of the initialization routine. You could write the entire initialization routine in assembler and not call a C function.

When control returns from the C function, the assembler procedure returns to the system. Note that this procedure must return with a FAR return instruction; such a return instruction is automatically generated by declaring ENTRYPOINT as a FAR procedure, as follows:

```
ENTRYPOINT PROC FAR
```

Figure 6-3

```
;               Figure 6-3
;               Module definition file for linking the dynamic-link module
;               of Figures 6-1 and 6-2.
;
LIBRARY FIG6_1 INITINSTANCE

PROTMODE

DATA MULTIPLE SHARED

EXPORTS
                DYNAFUNC  @1
```

Figure 6-4

```
#     Figure 6-4
#     MAKE file for preparing the dynamic-link module defined in Figures 6-1 and
#     6-2.
#
FIG6_1.OBJ : FIG6_1.ASM
     masm /MX FIG6_1.ASM;

FIG6_2.OBJ : FIG6_2.C
     cl /c /W2 /ASw /G2s /Zp FIG6_2.C

FIG6_1.DLL : FIG6_1.OBJ FIG6_2.OBJ FIG6_3.DEF
     link /NOI /NOD FIG6_1.OBJ FIG6_2.OBJ,FIG6_1.DLL,NUL.MAP,OS2.LIB,FIG6_3.DEF

FIG6_1.LIB : FIG6_3.DEF
     implib FIG6_1.LIB FIG6_3.DEF
```

The C initialization function is declared through the following line:

```
EXTRN _InitRoutine : FAR
```

The EXTRN keyword indicates that the function is defined in another source file, and the FAR keyword indicates that it must be called with a far call instruction. Also, InitRoutine must be defined in the C source file as a far function, so that it returns with a far return instruction. In the small-code models, InitRoutine could be declared and defined as a near function since it is contained within the same code segment as the procedure ENTRYPOINT; however, by explicitly defining it as far, you can use the same assembly language file regardless of the C memory model. The C compiler adds an underscore to the beginning of the function name, InitRoutine, when it writes it to the object file, since this routine is not declared as pascal.

As you will see in Chapter 7, you cannot install your own initialization routine when using the C runtime library within a dynamic-link module, since the library itself defines an initialization routine, and there can be only one initialization entry point in a given module.

See Chapter 12 for more information on using assembly language.

Write the C Initialization Function

Figure 6-2 contains the C source code for the example dynamic-link module. This file defines the main initialization function InitRoutine, which is called from the assembly language entry procedure. InitRoutine need not be defined with the _loadds keyword (or compiled with the /Au flag) since the system automatically loads the DS register with the selector of the module's automatic data segment, before it calls the initialization procedure.

InitRoutine first increments the process counter **ProcessCount**. Since this global data item is initialized to 0 and since the initialization routine is called once for each new client routine, ProcessCount maintains a count of the total number of processes that have referenced the dynamic-link library. ProcessCount does not necessarily contain the *current* number of active client processes, since it is not decremented when a client terminates. The example module presented in the second part of this chapter

adds a termination routine and maintains a count of the current number of active processes.

InitRoutine next prints one of two messages, depending upon whether it was called from the first client process or from a subsequent client process.

Finally, InitRoutine returns a value of 1 to indicate that the initialization routine was successful. When the assembly language initialization procedure (ENTRYPOINT) returns control to the system, it must supply a termination code in register AX. A nonzero value indicates that the function was successful, and a zero value indicates that the function failed. Conveniently, when a C function returns an integer value, it loads this value into register AX; accordingly, the C function can simply return the appropriate termination code and the assembler procedure need not explicitly load register AX. If you return 0 to indicate an error, the client program quietly dies; the OS/2 loader proceeds no further in running the program. As you will see in Chapter 8, returning 0 also causes termination of a client program that calls **DosLoadModule** to load the dynamic-link library at runtime.

Like a normal dynamic-link function, the initialization routine runs as part of the client process (specifically, the client process that just referenced the module). Accordingly, this routine is free to allocate file handles, memory segments, or other objects on behalf of the client. For example, you could add an initialization routine to the dynamic-link module presented in Chapter 4, which would open the printer device and obtain a valid printer handle for each new client process. Using such an initialization routine would eliminate the need for the functions in this module to test whether the printer has been opened.

By the time the initialization routine receives control, the system has allocated and initialized all global segments, as well as all instance data segments belonging to the current client. The initialization routine therefore has full access to these segments.

Specify When the Routine is to be Called

Notice that the LIBRARY statement in the module definition file of Figure 6-3 contains the keyword INITINSTANCE. This keyword is one of the

following two values that can be specified as the optional *initialization* field of the LIBRARY statement:

INITGLOBAL The initialization routine is called only for the *first* client process. This is the default condition.

INITINSTANCE The initialization routine is called for *all* client processes.

The example module is defined with the INITINSTANCE option since the initialization routine is to be called each time the module is referenced by a new client process. If you do not include either keyword, the initialization routine only will be called for the first client, since INITGLOBAL is the default.

If your initialization routine performs a set of initialization tasks that need to be performed only one time, and before any client program begins running, then you should specify the INITGLOBAL keyword, or simply do not specify an initialization value. For example, a subsystem that manages the printer may need to initialize the printer once only, before individual clients begin requesting printer services. Also, a screen management module may need to make a single determination of the current video configuration before client processes begin sending screen output.

If, however, the module must perform a set of initialization tasks for each client, or if it must keep track of the total number of client processes, you should specify the INITINSTANCE keyword. For example, the task of opening the printer in the module given in Chapter 4 would have to be performed for each client process, because a device handle is valid only within the process that obtained this handle from the system.

Typically, however, a module must perform both types of initialization tasks: certain tasks are performed only once, and other tasks are performed for each new client. You cannot install separate routines for each type of task. Nevertheless, you can easily perform both types of tasks within a single initialization routine that is called for each client (an INITINSTANCE routine) by maintaining a flag (in a global data segment) that indicates the first time the routine is called. For example, in the example initialization routine of Figure 6-2, the counter **ProcessCounter** has a value of 1 (after it is incremented at the beginning of the

routine) only for the first client. According to the value of this flag, the routine branches either to a routine that is performed only for the first client or to a routine that is performed for each subsequent client.

At what point during the processing of a client program is the initialization routine called? For a loadtime dynamic-link library, the initialization routine is called immediately before the client program receives control, as mentioned previously, it is called *after* the system has loaded and initialized all global segments and instance segments belonging to the client process. For a runtime dynamic-link library (discussed in Chapter 8), the client process explicitly loads the dynamic-link module through the DosLoadModule function. In this case, the initialization routine is called during the execution of DosLoadModule, which does not return control to the client until the initialization routine has completed.

Testing the Initialization Routine

Figure 6-4 provides a MAKE file for preparing the example dynamic-link library. The assembly language source file of Figure 6-1 is assembled using the **/MX** flag, which causes the assembler to preserve the case of public and external names. Normally, it converts these names to uppercase). Accordingly, the C function InitRoutine is written in the assembler file using uppercase and lowercase letters with a leading underscore.

Figure 6-5 lists a program you can use to test the dynamic-link module. This program simply calls the dynamic-link function **DynaFunc**, which displays a message and pauses for keyboard input. When you run the program, you will first see the message printed by the initialization routine, followed by the message from DynaFunc. If you run more than one instance of this program concurrently (by starting the program from two or more screen groups as described in Chapter 5) you will see that the initialization routine is called each time the program is loaded.

You can compile and link this program using the following two commands:

```
cl /c /W2 /G2 /Zp FIG6_5.C
```

and:

```
link /NOI /NOD FIG6_5.OBJ,, NUL.LST, FIG6_1.LIB SLIBCE.LIB OS2.LIB;
```

Figure 6-5

```
/*
     Figure 6-5
     A C program for testing the example dynamic-link modules given in
     Chapter 6.
*/

void pascal far DynaFunc (void);          /* Defined in dynamic-link module.   */

void main (void)
     {
     DynaFunc ();                         /* Dynamic-link function.            */

     } /* end main */
```

Writing Termination Routines

Figures 6-6 through 6-9 list a set of files for building an example dynamic-link library that has a termination routine in addition to an initialization routine.

The assembly language file of Figure 6-7 defines the initialization entry point in exactly the same manner as the listing of Figure 6-1, explained in the previous section. As in the prior example, the body of the initialization routine is contained in the C function InitRoutine, defined in Figure 6-7. This function first increments the process counter, ProcessCount, and then calls the API function **DosExitList** to install the termination routine.

Figure 6-6

```
;     Figure 6-6
;     This file defines the initialization entry point for the dynamic-link
;     module of Figures 6-6 and 6-7.
;
.MODEL LARGE
.CODE

EXTRN _InitRoutine : FAR            ;Initialization routine defined in the C
                                    ;source file of Figure 6-7.

ENTRYPOINT PROC FAR

     call _InitRoutine              ;Call main initialization routine.
     ret                            ;FAR return to the system.

ENTRYPOINT ENDP

END ENTRYPOINT                      ;Defines the initialization entry point.
```

Figure 6-7

```
/*
     Figure 6-7
     This C source file defines the initialization routine 'InitRoutine',
     the termination routine 'TermRoutine' and the dynamic-link function
     'DynaFunc'.
*/

#define INCL_SUB
```

```
#define INCL_DOSPROCESS
#include <OS2.H>

/*** Global data: ************************************************************/

USHORT far ProcessCount = 0;  /* Maintains a count of the current number of  */
                              /* active client processes.                    */

/*** Instance data: **********************************************************/

char *Message1;               /* Pointer to first message displayed by       */
                              /* 'TermRoutine'.                              */

                              /* Second message displayed by 'TermRoutine':  */
char Message2 [] = "|         remaining processes:              | \r\n";

int _acrtused = 0;            /* Define variable to avoid linking in C       */
                              /* startup code.                               */

/*** Termination function: ***************************************************/

void pascal far TermRoutine (USHORT TermCode);

/*** Private function: *******************************************************/

void UShortToString (USHORT Source, char *Target);  /* Converts an unsigned  */
                                                    /* integer to a string.  */

/*** Initialization routine: *************************************************/
```

```
int far InitRoutine
     (void)
     {
     ++ProcessCount;              /* Increment count of active clients.     */

     DosExitList                  /* Install the termination routine.       */
          (EXLST_ADD,             /* Code for desired service: add a routine*/
          TermRoutine);           /* Address of termination routine to add. */

     return (1);                  /* Return value of 1 indicates that the   */
                                  /* initialization routine was successful. */

     } /* end InitRoutine */

/*** Dynamic-link function: ************************************************/

void pascal far DynaFunc (void)
     {
     KBDKEYINFO Key;                    /* Structure to receive keyboard key.*/

     VioWrtTTY ("\r\n", 48, 0);
     VioWrtTTY ("                                                  \r\n", 48, 0);
     VioWrtTTY ("                                                  \r\n", 48, 0);
     VioWrtTTY ("       Dynamic-link function called.              \r\n", 48, 0);
     VioWrtTTY ("                                                  \r\n", 48, 0);
     VioWrtTTY ("                                                  \r\n", 48, 0);
     VioWrtTTY ("         press any key to continue                \r\n", 48, 0);
     VioWrtTTY ("\r\n", 48, 0);
```

```
    KbdCharIn (&Key, IO_WAIT, 0);       /* Pause for keyboard input.          */

    } /* end DynaFunc */

/*** Termination routine: ************************************************/

void pascal far TermRoutine (USHORT TermCode)
    {
    --ProcessCount;                     /* Decrement count of active clients.*/

    switch (TermCode)                   /* Assign string to 'Message1' based */
                                        /* on cause of process termination.  */
        {
        case TC_EXIT:
            Message1 = " |               normal exit                | \r\n";
            break;

        case TC_HARDERROR:
            Message1 = " |           critical error abort            | \r\n";
            break;

        case TC_TRAP:
            Message1 = " |              trap operation               | \r\n";
            break;

        case TC_KILLPROCESS:
            Message1 = " |process terminated through DosKillProcess | \r\n";
```

```
            break;
        default:
            Message1 = " |       unknown cause of termination       | \r\n";
            break;

        }
                                    /* Write the number of remaining     */
                                    /* client processes to 'Message2':   */
    UShortToString (ProcessCount, Message2 + 32);

                                    /* Print the message.                */
    VioWrtTTY ("\r\n", 48, 0);
    VioWrtTTY ("                Termination Routine                \r\n", 48, 0);
    VioWrtTTY ("                                                   \r\n", 48, 0);
    VioWrtTTY (Message1, 48, 0);
    VioWrtTTY (Message2, 48, 0);
    VioWrtTTY ("                                                   \r\n", 48, 0);
    VioWrtTTY ("                                                   \r\n", 48, 0);
    VioWrtTTY ("\r\n", 48, 0);

    DosExitList                     /* NOTIFY THE SYSTEM THAT TERMINATION*/
        (EXLST_EXIT,                /* PROCESSING IS COMPLETE.           */
        0);

    } /* TermRoutine */


/*** Private function: ****************************************************/

void UShortToString (USHORT Source, char *Target)
/*
    This function converts the USHORT value 'Source' to a string, which is
    written to the buffer 'Target'.  The resulting string is NOT NULL
    terminated.
```

```
*/
     {
     register int i = 0;
     char Temp [6];

     do
          {
          Temp [i++] = (char)(Source % 10 + '0');
          Source /= 10;
          }
     while (Source);

     do
          {
          *Target++ = Temp [--i];
          }
     while (i);

     } /* end UShortToString */
```

Figure 6-8

```
;              Figure 6-8
;              Module definition file for linking the example
;              dynamic-link module of Figures 6-6 and 6-7.
;
LIBRARY FIG6_6 INITINSTANCE

PROTMODE

DATA MULTIPLE SHARED

EXPORTS
     DYNAFUNC  @1
```

Figure 6-9

```
#     Figure 6-9
#     MAKE script for preparing the dynamic-link module defined in Figures
#     6-6 and 6-7.
#
FIG6_6.OBJ : FIG6_6.ASM
     masm /MX FIG6_6.ASM;

FIG6_7.OBJ : FIG6_7.C
     cl /c /W2 /ASu /G2s /Zp FIG6_7.C

FIG6_6.DLL : FIG6_6.OBJ FIG6_7.OBJ FIG6_8.DEF
     link /NOI /NOD FIG6_6.OBJ FIG6_7.OBJ,FIG6_6.DLL,NUL.MAP,OS2.LIB,FIG6_8.DEF

FIG6_6.LIB : FIG6_8.DEF
     implib FIG6_6.LIB FIG6_8.DEF
```

The **DosExitList** function is summarized in Figure 6-10. You should call this function from an initialization routine that is invoked for each client process (an INITINSTANCE routine), although you could also call it the first time a dynamic-link function is called by a new client. DosExitList installs the specified termination routine for the current client process; therefore, you must call this function from each client process.

The initialization routine calls DosExitList as follows:

```
DosExitList
     (EXLST_ADD,
     TermRoutine);
```

Figure 6-10: The DosExitList Function.

DosExitList

☐ *Purpose:* Installs or removes a function from the list of routines that are automatically executed when the current process terminates; it is also called by a termination routine to notify the system that it has completed its termination processing.

☐ *Prototype:* USHORT APIENTRY DosExitList

(USHORT fFnCode — One of the following codes indicating the desired service:

Code	Desired service
EXLST_ADD	Add function to termination list.
EXLST_REMOVE	Remove function from termination list.
EXLST_EXIT	Call next function on termination list (this code is supplied by a termination routine to indicate that it has completed its termination processing).

PFNEXITLIST pfnFunction — Address of the function to add or remove from the termination list (if DosExitList is called by the termination routine itself to conclude processing, this value should be 0). The definition of the type PFNEXITLIST is given below.

☐ *Type Definition*: typedef VOID (PASCAL FAR *PFNEXITLIST)(USHORT);

☐ *Return Value:* If successful, the function returns zero. If an error occurs, it returns a nonzero error code. The following are the possible error codes:

ERROR_INVALID_DATA	The function call specified an invalid value for the first parameter (fFnCode).
ERROR_NOT_ENOUGH_MEMORY	Insufficient free memory to add exit routine

The value passed as the first parameter, EXLST_ADD, indicates that the specified routine should be added to the list of termination routines for the current process. The second parameter, **TermRoutine**, gives the

address of this routine. The system maintains a list of termination routines for a given process. The client program can also install one or more termination routines. When the current process terminates—for any reason—the routines on this list are called one at a time, in an unpredictable order.

The termination routine for the example module is the function **TermRoutine**, defined in the C source code of Figure 6-7. This function first decrements the process counter, ProcessCount. ProcessCount is initialized to 0; it is then incremented by the initialization routine each time a new client process begins, and it is decremented by the termination routine each time a client process ends. Thus, it maintains a count of the current number of active client processes. This variable must, of course, be stored in a global data segment.

The system passes the termination routine a single parameter, which contains a code indicating the cause for the process termination. Table 6-1 lists the possible codes:

Table 6-1: Parameter Values and Termination Causes

PARAMETER VALUE	TERMINATION CAUSE
TC_EXIT	Normal exit (the process called DosExit).
TC_HARDERROR	Process was aborted through a critical-error handler.
TC_TRAP	Process was terminated through a trap operation.
TC_KILLPROCESS	Process was terminated due to another process calling DosKillProcess.

TermRoutine prints a message that indicates the reason the process terminated, and displays the number of active client processes remaining.

Finally, TermRoutine calls the DosExitList function, passing the value EXLST_EXIT as the first parameter, which notifies the system that the termination routine is complete. It passes a 0 as the second parameter, since it is not installing or removing a termination routine. The termina-

tion routine must call DosExitList to allow the system to continue terminating the process. The routine must not issue a simple return instruction. The system passes control to the termination routine through a JMP instruction rather than a CALL instruction; therefore, the stack does not contain the return address required for a RET instruction.

Note that TermRoutine is declared as follows:

```
void pascal far TermRoutine (USHORT TermCode);
```

This declaration is necessary to conformto the type of the second parameter passed to DosExitList, as it is declared in the OS/2 header files.

Like an initialization routine or a normal dynamic-link function, the termination routine runs as part of the client process. The module can take advantage of the termination routine to release any objects held for the process in global data segments (such as data buffers, semaphores, or lists of pending requests), and to perform any other required final duties. The system automatically releases all instance segments belonging to the process. The system also automatically frees all file handles, memory blocks, or other objects dynamically allocated by the process. The termination routine, however, should flush any data buffers and properly close any files it has opened for the client process to make sure that all file data preserved.

When the termination routine receives control, the process is already in a state of partial termination. For example, all process threads except the one executing the termination routine have been ended. By the time the termination routine is invoked, it is too late to resume the process. The termination routine is not the appropriate point in the code to print a statement such as "Are you sure you want to end the program?" Also, you should not attempt to start new threads or processes from the termination routine by calling **DosCreateThread** or **DosExecPgm**.

Finally, since the system cannot complete terminating the process until the termination routine finishes, this routine should be as short as possible. Usually, when the termination routine receives control, the user is attempting to exit the program and is in no mood for a long delay.

The OS/2 function **DosSetSigHandler** allows you to install a routine that is called under certain termination conditions. This function installs a handler for one of several types of signals that may be sent to the process;

certain exit conditions, such as the user pressing Ctrl-C, result in specific signals that can be processed by a signal handler. You should not, however, use this function to install a general termination routine for a dynamic-link library, for two reasons. First, the routine you install is not called for a normal process termination, which occurs when the process calls DosExit—the most common means of ending a program. Second, unlike DosExitList, DosSetSigHandler installs only a single routine per process for a given type of signal; therefore, the routine you install would be removed if the client later installs a routine for the same signal.

Testing the Termination Routine

You can test the termination routine using the client program listed in Figure 6-5 which was employed in the previous section to test the initialization routine. You can compile and link this program using the following two command line instructions:

```
cl /c /W2 /G2 /Zp FIG6_5.C
```

and:

```
link /NOI /NOD FIG6_5.OBJ,, NUL.LST, FIG6_6.LIB SLIBCE.LIB OS2.LIB;
```

When you run the program, you will see the message printed by the termination routine immediately before the program returns to OS/2. End the program both by pressing a key, and by typing Control-C, and notice the different termination messages. By using separate screen groups, you can also run multiple instances of the programs to test the counter of client processes maintained by the dynamic-link module.

CHAPTER 7

USING THE C RUNTIME LIBRARY

Calls to functions in the C runtime library have been absent from many of the example programs and all of the dynamic-link modules presented so far in this book. As you have seen, the standard C runtime library has two major limitations. First, most of the C library functions cannot be called by more than one simultaneous thread within a single process and, therefore, the C runtime library generally cannot be used by multiple-thread programs or dynamic-link modules. Second, none of the C library functions can be called from dynamic-link modules, even if the module code is executed by only a single thread per process. Consequently, the example multiple-thread programs and dynamic-link libraries given in prior chapters have supplied their own supporting functions (such as _StrLen in Figure 4-2 and UShortToString in Figure 5-17).

However, Microsoft C (beginning with version 5.1) supplies three special versions of the runtime library that overcome these limitations. Each of the three special runtime library versions is designed for use within a specific programming environment; these library versions may be summarized as follows:

1. A version for multiple-thread application programs (in the file LLIBCMT.LIB).

2. A version for single-thread dynamic-link libraries (in the file LLIBCDLL.LIB).

3. A version for *both* multiple-thread dynamic-link libraries *and* multiple-thread applications (in the file CRTLIB.DLL).

This chapter describes each of these libraries and presents the techniques for using them in your programs and dynamic-link modules.

Calling your favorite C runtime functions does not come without a cost. This chapter emphasizes the restrictions that apply when using each of the special library versions, and its conclusion explains why you might not want to use the special versions of the C runtime library when developing a general purpose dynamic-link library. The chapter emphasizes basic techniques and general considerations for selecting and using special versions of the C runtime library. As in the other chapters, specific details and examples are based upon Microsoft C, version 5.1. Be sure to consult your compiler documentation for the exact names of the various files and for additional programming details, especially if you are using a different compiler or compiler version.

Multiple-Thread Applications

One special version of the C runtime library supplied by Microsoft C (5.1) is contained in the library file LLIBCMT.LIB. This library version has the following four basic properties:

1. The functions in this library can be called only by application programs; they cannot be called by dynamic- link libraries.

2. The library is statically linked to your program.

3. The functions can be called by multiple threads within a single process.

4. Only a large memory model version is supplied.

The remainder of this section elaborates on these four features and illustrates them through an example program, which is listed in Figure 7-1. This program, like the example application in Figure 1-8, begins several new program threads; each of the newly started threads repeated-

ly prints a message on the screen, while the original thread waits for a keystroke and terminates the entire process when a key is pressed. Unlike the program of Figure 1-8, however, the example program is linked with a special C library version (LLIBCMT.LIB) and freely calls C library functions. Figure 7-2 provides a MAKE script for preparing this program.

Figure 7-1

```
/*
     Figure 7-1

     This program demonstrates using the LLIBCMT.LIB C runtime library, which:

     o  is statically linked
     o  supports multiple-thread application programs
     o  can be called only by an application program;  cannot be called by a
        dynamic-link library

*/

#define INCL_DOSPROCESS                         /* Definition of 'DosSleep'.    */
#include <OS/2.H>

#include <MT\STDIO.H>                           /* Include special multi-       */
#include <MT\CONIO.H>                           /* tasking versions of the C    */
#include <MT\PROCESS.H>                         /* header files.                */
#include <MT\STDDEF.H>                          /* Defines _threadid.           */

char StackArea [6144];                          /* Space for 3 2K stacks.       */

void far NewThread (int far *Argument);         /* Function executed by new     */
                                                /* threads.                     */
```

```
void main (void)
     {
     int NewThreadCount = 0;              /* Count of new threads started.     */
     char far *StackBase = StackArea;     /* Points to base of each new        */
                                          /* thread's stack.                   */
     int ThreadID;                        /* Holds thread ID returned by       */
                                          /* '_beginthread'.                   */

     do
          {
          ++NewThreadCount;               /* Increment count of new threads.   */

          ThreadID = _beginthread         /* Start a new thread.               */
               (NewThread,                /* Address of function to be         */
                                          /* executed by new thread.           */
               StackBase,                 /* Base of new thread's stack.       */
               2048,                      /* Size of new thread's stack.       */
               &NewThreadCount);          /* Argument passed to new thread:    */
                                          /* address of new thread count.      */

          DosSleep (500L);                /* 1/2 second pause between starting */
                                          /* new threads.                      */

          if (ThreadID == -1)             /* ID value of -1 indicates an error.*/
               {
               printf ("Error beginning new thread number %d\n",
                       NewThreadCount);
               exit (1);
               }

          StackBase += 2048;              /* Move pointer to next stack area.  */
```

```
        }
    while (NewThreadCount  3);

    getch ();                          /* Pause for keyboard input.          */

    } /* end main */

void far NewThread (int far *Argument)
    {
    int ThreadID = *_threadid;         /* Obtain ID of new thread from       */
                                       /* global pointer '_threadid'.        */
    for (;;)
        {
        printf ("Hello from new thread ID number %d.  Number of new "
                "threads started:  %d\n", ThreadID, *Argument);

        DosSleep (500L);               /* Pause after printing message --    */
                                       /* and yield remainder of time slice.*/
        }

    } /* end NewThread */
```

Figure 7-2

```
#    Figure 7-2
#    A MAKE file for preparing the example program of Figure 7-1
#
FIG7_1.OBJ : FIG7_1.C
    cl /c /W2 /ALw /G2 /Zp FIG7_1.C

FIG7_1.EXE : FIG7_1.OBJ
    link /NOI /NOD FIG7_1.OBJ,, NUL.LST, LLIBCMT.LIB OS2.LIB;
```

The main feature of the LLIBCMT library version is that it can be used only by an application program and not by a dynamic-link module. The other two library versions described in the chapter are designed to be used by dynamic-link modules.

The second feature is that the library is statically (that is, conventionally) linked to the application in the same manner as the standard C runtime library. When linking the program, you simply specify the library name LLMIBMT.LIB rather than the standard library name (for example, LLIBCE.LIB). Since the compiler normally writes the name of the standard library to the object file as a default search library, you must be careful that you do not accidently link the program with a standard version of the library. Accordingly, the MAKE script of Figure 7-2 includes the **/NOD** switch, which prevents default library searches. Also, even if your program makes no explicit calls to the OS/2 API, you must specify the import library OS2.LIB (DOSCALLS.LIB on some systems) to resolve the references to OS/2 API functions made by the C library.

The third significant feature allows you to call a given function in this C library version from more than one simultaneous program thread. Remember from Chapter 1 that most of the standard C library functions are **nonreentrant**—they can be called by only one program thread at a time—simultaneous instances of such functions will tend to corrupt each other's data. The functions in the LLIBCMT library, however, are **reentrant**; accordingly, you can make full use of these C functions within a multiple-thread application.

The example program of Figure 7-1 takes advantage of this freedom by calling the C library function **printf** to print formatted messages from each of the newly started threads, and by calling the C function **getch** to pause for keyboard input within the main function, executed by the original program thread.

When using a version of the C library that supports multiple-thread programs—the versions discussed in this section and in the last section in the chapter—the program must manage threads by using the special C library functions **_beginthread** and **_endthread**, rather than the standard OS/2 API functions **DosCreateThread** and **DosExit**. DosCreateThread and DosExit were used by the example multiple-thread applications presented in Chapter 1. If you use these functions in a

program that employs a special version of the C runtime library, the results are unpredictable.

The C function _beginthread starts a new program thread; it differs from DosCreateThread in two primary ways. First, you can supply _beginthread a value (the fourth parameter) that it will pass as a parameter to the function executed by the new thread. This parameter has the size of a far pointer, and you can use it to pass a simple numeric value, or the address of a string or other data item.

The example program begins three new threads; each of these threads executes the same function, **NewThread** (thus creating three instances of a single function). NewThread is declared as follows:

```
void far NewThread (int far *Argument);
```

The parameter **Argument** is used to pass each of the new threads the far address of the variable *NewThreadCount*, which contains a count of the total number of new threads started by the function main. The threads print this value as part of the messages they send to the screen; as you will see if you run the program, the value begins at 1 and quickly reaches 3, where it remains.

The parameter passed to the function executed by the new thread must be declared as a far pointer to some data type in order to conform to the declaration of the _beginthread function. However, if you want to pass a simple numeric type, such as an unsigned integer, you can cast the parameter to the appropriate type, as in the following code fragment:

```
void far NewThread (int far *Argument)
     {
     unsigned LocalUInt;
     .
     .
     .
     LocalUInt = (unsigned)Argument;
```

The second difference between DosCreateThread and _beginthread is that DosCreateThread is passed only the address of the top of the stack reserved for the new thread (it does not need to know how large the stack is). The function _beginthread, however, must be passed the address of the bottom of the stack and the size of the stack. The C library uses the stack size information to provide stack checking for each program thread, just as the standard C library provides a stack check routine that is called by the functions of a single-thread program. However, as you will see later in this section, you must disable stack checking if you compile the program under the small or compact memory model.

The example program reserves a 2-kilobyte stack area for each new thread. If a thread executes one or more C runtime functions, it should have a stack of at least 2 kilobytes. If it calls one or more OS/2 API functions, its stack should be larger. The example program in Chapter 1 reserves a 3-kilobyte stack for each thread, since these threads invoke API functions.

The C library function _endthread terminates a thread begun by _beginthread (calling _endthread is analogous to calling DosExit, passing it the value EXIT_THREAD). The _endthread function is not normally required, since the thread will automatically be terminated if it returns from the function it initially executed. The function _endthread would be useful, however, for summarily aborting the current thread from within a nested subroutine.

See your compiler documentation for the prototypes of these two functions and for further information on their use.

The C runtime library supplies a global variable, *_threadid* (defined in STDDEF.H), which is a far pointer holding the address of a location in memory containing the identifier of the thread that is currently executing. This location is within the local information segment, which is maintained by OS/2 and can also be accessed through the DosGetInfoSeg function. The new threads in the example program print the identifier of the current thread, which is obtained from the expression ***_threadid**.

Finally, because the program employs multiple threads, you must compile it using the /Aw flag as explained in Chapter 1. When you use this flag remember to specify the memory model; since the example

program is compiled under the large memory model, the MAKE file of Figure 7-1 specifies the flag **/ALw**.

A final important feature of the LLIBCMT library is that only a large memory model version is provided. The advantage of a large memory-model library is that—with care—its functions can be called from any memory model program.

A set of special header files facilitates calling the functions in this library from programs compiled under memory models other than the large or huge model. You must use these header files when employing either of the two versions of the C runtime library that support multiple-thread code (that is, the library versions discussed in this section and in the last section of the chapter). These header files have the same names as the standard C header files (such as STDIO.H and STDDEF.H), but are normally placed in the MT subdirectory of the INCLUDE directory. Accordingly, when including C header files, you must preface the header filenames with the subdirectory designation. For example, the following statement includes the multiple-thread version of the standard I/O header file:

```
#include  <\MT\STDIO.H>
```

Among the unique features of these header files are function declarations using the far keyword; all functions—as well as all address parameters passed to these functions—are explicitly declared as far. Accordingly, when calling the functions from a small, compact, or medium memory model program, the compiler will automatically perform most of the required type conversions. However, if the function returns an address, you must receive it in a far pointer. For example, if you call **fopen** from a small or medium model program, the variable that receives the FILE pointer must be explicitly declared far, as in the following code:

```
FILE far *fp
.
.
.
fp = fopen ("PRN","w");
```

Also, consider the following function call:

```
printf ("hello, %s\n","world");
```

It is left for the reader to puzzle why this call to a function in the LLIBCMT version of the C library causes a protection fault if the program is compiled under the small or medium memory model (even if you have included the special header file MT\STDIO.H).

Finally, if you use the small or compact memory model, you must disable stack checking (with the /Gs flag), since stack checking performed by the LLIBCMT library requires a memory model that uses far function calls. It is best to simply compile the program under the large memory model. Accordingly, the MAKE file of Figure 7-2 specifies the large model.

Single-Thread Dynamic-Link Libraries

A second special version of the runtime library provided by Microsoft C is contained in the file LLIBCDLL.LIB. It has the following features:

1. The functions in this library can be called only by dynamic-link libraries; the library is not designed to be used by application programs.

2. The library is statically linked to the dynamic- link module.

3. The library functions can be called by only one simultaneous program thread; the library is not designed for multiple-thread code.

4. Only a large memory model version is supplied.

Figures 7-3 through 7-6 demonstrate the use of this version of the C library. Figure 7-3 defines the dynamic-link function **PrintMessage**, which prints a message on the screen using the C library function printf. Figure 7-5 lists a client program that first prints its own message and then calls PrintMessage to print a message from the dynamic-link library. Note that the MAKE file of Figure 7-6 prepares both the dynamic-link module and the client program.

Figure 7-3

```
/*
     Figure 7-3
     This is the source file for a dynamic-link library that demonstrates
     using the LLIBCDLL.LIB C runtime library, which:

     o  is statically linked

     o  can be called by only one thread at a given time

     o  is designed to be called from a dynamic-link library

*/

#include <STDIO.H>        /* Include single-tasking version of C header file. */

void pascal far _loadds PrintMessage
     (void)
     {
     printf ("Hello from the dynamic-link library.\n");

     } /* end PrintMessage */
```

Figure 7-4

```
;              Figure 7-4
;              A module definition file for linking the dynamic-link module
;              of Figure 7-3.
;
LIBRARY FIG7_3

PROTMODE

DATA MULTIPLE

EXPORTS
     PRINTMESSAGE    @1
```

Figure 7-5

```
/*
     Figure 7-5

     This program calls the dynamic-link library function 'PrintMessage',
     defined in Figure 7-3.  The program is linked with the standard (single-
     tasking, nondynamic-linked) version of the C runtime library.
*/

#include <STDIO.H>              /* Include single-tasking header file.       */

void pascal far _loadds PrintMessage (void);

void main (void)
     {
     printf ("Hello from the program.\n");

     PrintMessage ();           /* Call the dynamic-link function.           */

     } /* end main */
```

The first feature of the LLIBCDLL library is that it is designed for use only by dynamic-link modules. Consequently, client programs must use another suitable runtime library version. The example dynamic-link library uses the LLIBCDLL library, while the example client program uses the standard C library.

The second feature of this version of the C library is that it is statically linked to the calling dynamic-link module. Consequently, when linking this module, you should specify the library file LLIBCDLL.LIB, and use the /NOD option so that no other library version is inadvertently linked. You must also specify the OS2.LIB (or DOSCALLS.LIB) import library to

resolve the references to OS/2 API functions made by the C library, and possibly by your code.

Since the dynamic-link module and the C library code are statically linked together, they share the same automatic data segment. As in previous examples, you should define the automatic data segment as an instance segment (using the DATA MULTIPLE statement in the module definition file) so that each process has its own private copy of this segment.

Figure 7-6

```
#    Figure 7-6
#    A MAKE file for preparing the dynamic-link module of Figure 7-3, and the
#    client program of Figure 7-5.
#
FIG7_3.OBJ : FIG7_3.C
     cl /c /W2 /ALw /G2s /Zp FIG7_3.C

FIG7_3.DLL : FIG7_3.OBJ FIG7_4.DEF
     link /NOI /NOD FIG7_3.OBJ, FIG7_3.DLL,, LLIBCDLL.LIB OS2.LIB, FIG7_4.DEF

FIG7_3.LIB : FIG7_4.DEF
     implib FIG7_3.LIB FIG7_4.DEF

FIG7_5.OBJ : FIG7_5.C
     cl /c /W2 /G2 /Zp FIG7_5.C

FIG7_5.EXE : FIG7_5.OBJ FIG7_3.LIB
     link /NOI /NOD FIG7_5.OBJ,, NUL.LST, FIG7_3.LIB SLIBCE.LIB OS2.LIB;
```

If a dynamic-link module uses a special version of the C runtime library (either the version discussed in this section or the one discussed in the next section) you must link the module with the C startup code—unlike the dynamic-link modules presented in previous chapters, which did not use the C library. Consequently, you should not define the variable *_acrtused* within the source file. If you do not define *_acrtused*, the linker will automatically extract the startup code from the C library file, and insert it into the resulting dynamic-link library.

The beginning of the C startup code is defined as the initialization entry point. The object module containing the C startup routine includes an END statement that specifies this routine as the module entry point. Consequently, the C startup code is automatically executed when the dynamic-link module is first referenced by an application program. In the example dynamic-link modules supplied with the Microsoft C compiler, the initialization routine is defined to be global. It is called only when the module is referenced by the first client program; this option is selected by leaving the initialization field of the LIBRARY statement in the module definition file blank. The module definition file given in Figure 7-4 likewise defines the initialization routine as global.

As mentioned in Chapter 6, a dynamic-link module can define only one initialization entry point. Consequently, if you use a special version of the C runtime library within your dynamic-link module, you cannot define your own initialization routine. If you define an initialization routine using the END statement as described in Chapter 6, this routine will replace the routine provided by the C library; C library functions will subsequently fail, since this library requires its own initialization routine.

A third feature of the LLIBCDLL C runtime library is that the functions can generally be called by only a single process thread at a given time. The dynamic-link module, therefore, cannot run multiple threads that call C library functions. Also, a client program must not call the dynamic-link functions from more than one simultaneous program thread. Accordingly, this version of the C library places a restriction not only on the dynamic-link module itself, but also on all client programs that use the module. The C library version described in the next section removes this restriction.

You can call the LLIBCDLL library from multiple-thread code if you explicitly serialize access to the C functions by using semaphores. Figure 7-7 lists a version of the dynamic-link module given in Figure 7-3, which uses semaphores to serialize access to the C runtime library function printf. This module version, therefore, could be called from more than one simultaneous thread within a client program. The section on Adding Interprocess Communication in Chapter 1 describes how to use semaphores for this purpose.

Surrounding calls to C functions with invocations to semaphore management functions is a clumsy solution. A better method is to use the multiple-thread C library version described in the next section.

A fourth feature of the LLIBCDLL library is that, like the LLIBCMT library described in the previous section, only a large memory model version is provided. Since the LLIBCDLL library is a single-thread library version, you must use the standard header files rather than the special multiple-thread header files (those in the MT subdirectory). Since the declarations in the standard header files do not explicitly make all functions and address parameters far, the compiler will not perform the required type conversions when calling this library from small, medium, or compact memory model programs. Consequently, you should use the large (or huge) memory model when employing this library version.

Multiple-Thread DLLs and Applications

The third and final version of the C runtime library is contained in the files CRTLIB.DLL (the library code itself) and CRTLIB.LIB (the import library). The CRTLIB version of the library has the following features:

1. The functions in this library can be used by both application programs and dynamic-link modules.

2. The library is dynamically-linked to the program or dynamic-link module.

3. The library functions can be called by multiple threads within a single process.

4. Only a large memory model version is supplied.

Figure 7-7

```
/*
     Figure 7-7
     This is the source file for a dynamic-link library that demonstrates
     using the LLIBCDLL.LIB C runtime library, which:

     o  is statically linked
     o  can be called by only one thread at a given time
     o  is designed to be called from a dynamic-link library

     This source file is the same as that given in Figure 7-3, except that it
     uses a semaphore to serialize access to the CRT library, allowing the
     function 'PrintMessage' to be called by more than one thread within a
     single process at given time.
*/

#define INCL_DOSSEMAPHORES
#include <OS2.H>

#include <STDIO.H>          /* Include single-tasking version of C header file. */

ULONG CRTSem = 0;           /* Semaphore for serializing access to the CRT      */
                            /* library.                                         */

void pascal far _loadds PrintMessage
     (void)
     {
     DosSemRequest          /* Wait for semaphore to clear and then set it.     */
          (&CRTSem,         /* Semaphore handle (its far address).              */
          -1L);             /* Wait flag:  -1 means wait forever.               */
```

```
printf ("Hello from the dynamic-link library.\n");

DosSemClear          /* Clear the semaphore.                    */
     (&CRTSem);      /* Semaphore handle.                       */

} /* end PrintMessage */
```

Figures 7-8 through 7-11 demonstrate the use of the CRTLIB library. The dynamic-link module source code in Figure 7-8 defines the function PrintMessage, which uses the C function printf to print a message on the screen. PrintMessage is similar to the function defined in the dynamic-link module of Figure 7-3. Because the module uses the CRTLIB version of the C runtime library it can be called with impunity from a multiple-thread client program. To demonstrate this ability, Figure 7-10 lists a multiple-thread program. This program is the same as the one in Figure 7-11, except that rather than directly calling printf from each new thread, it calls the dynamic-link function PrintMessage (which then calls printf).

The first feature of the CRTLIB C library version is that it can be used by both normal application programs and by dynamic-link libraries. Since this C library version is contained in a dynamic-link library file any application program can use this library and benefit from the advantages of the dynamic linking mechanism. If a dynamic-link module uses the CRTLIB library, however, all of its client programs must also use the CRTLIB version of the C runtime library. Unfortunately, a client program is not free to select another runtime library version.

The CRTLIB runtime library is also packaged as a dynamic-link library and is therefore dynamically linked to your programs and dynamic-link modules. Thus, the C library functions are called and linked in the same manner as the functions of the OS/2 API. Figure 7-12 on page 221 illustrates the calling relationships among the dynamic-link module that you write, the client programs that use this module, and the CRTLIB dynamic-link library file shared by the module and all its clients.

Figure 7-8

```
/*
     Figure 7-8
     This is the source file for a dynamic-link library that demonstrates
     using the C runtime library contained in the files CRTLIB.DLL (the
     function code) and CRTLIB.LIB (the import library).  This library version:

     o  is dynamically linked
     o  can be called by multiple simultaneous threads within a single process
     o  must be used by both the dynamic-link library and the client program
*/

#define DLL              /* Indicate dynamic-link version of CRT library.    */
#include <MT\STDIO.H>    /* Include multitasking version of C header file.   */

void pascal far _loadds PrintMessage
     (int ThreadID, int NumThread)
     {
     printf ("Hello from new thread ID number %d.  Number of new "
             "threads started:  %d\n", ThreadID, NumThread);

     } /* end PrintMessage */
```

Figure 7-9

```
;    Figure 7-9
;            A module definition file for linking the dynamic-link
;            module of Figure 7-8.
;
```

```
LIBRARY FIG7_8

PROTMODE

DATA MULTIPLE

EXPORTS
     PRINTMESSAGE   @1
```

Figure 7-10

```
/*
     Figure 7-10

     This program calls the multiple-thread dynamic-link function
     'PrintMessage', defined in Figure 7-8.  The program uses the dynamically-
     linked version of the CRT library, contained in the files CRTLIB.DLL (the
     function code) and CRTLIB.LIB (the import library).
*/

#define INCL_DOSPROCESS                 /* Definition of 'DosSleep'.    */
#include <OS2.H>

#define DLL                             /* Indicate dynamic-link CRT    */
                                        /* library is used.             */
#include <MT\STDIO.H>                   /* Include special multi-       */
#include <MT\CONIO.H>                   /* tasking versions of the C    */
#include <MT\PROCESS.H>                 /* header files.                */
```

```
#include <MT\STDDEF.H>                          /* Defines _threadid.          */

void pascal far _loadds PrintMessage            /* Dynamic-link function       */
     (int ThreadID, int NumThread);             /* defined in Figure 7-8.      */

char StackArea [6144];                          /* Space for 3 2K stacks.      */

void far NewThread (int far *Argument);         /* Function executed by new    */
                                                /* threads.                    */

void main (void)
     {
     int NewThreadCount = 0;                    /* Count of new threads started.    */
     char far *StackBase = StackArea;           /* Points to base of each new       */
                                                /* thread's stack.                  */
     int ThreadID;                              /* Holds thread ID returned by      */
                                                /* '_beginthread'.                  */

     do
          {
          ++NewThreadCount;                     /* Increment count of new threads.  */

          ThreadID = _beginthread               /* Start a new thread.              */
               (NewThread,                      /* Address of function to be        */
                                                /* executed by new thread.          */
               StackBase,                       /* Base of new thread's stack.      */
               2048,                            /* Size of new thread's stack.      */
               &NewThreadCount);                /* Argument passed to new thread:   */
                                                /* address of new thread count.     */
          DosSleep (500L);                      /* 1/2 second pause between starting */
                                                /* new threads.                     */
```

```
        if (ThreadID == -1)           /* ID value of -1 indicates an error.*/
             {
             printf ("Error beginning new thread number %d\n",
                     NewThreadCount);
             exit (1);
             }

        StackBase += 2048;            /* Move pointer to next stack area.  */


        }
    while (NewThreadCount  3);

    getch ();                         /* Pause for keyboard input.         */

    } /* end main */


void far NewThread (int far *Argument)
    {
    for (;;)
        {
        PrintMessage (*_threadid, *Argument);   /* Call the dynamic-link   */
                                                /* function defined in     */
                                                /* Figure 7-8.             */

        DosSleep (500L);              /* Pause after printing message --   */
                                      /* and yield remainder of time slice.*/
        }

    } /* end NewThread */
```

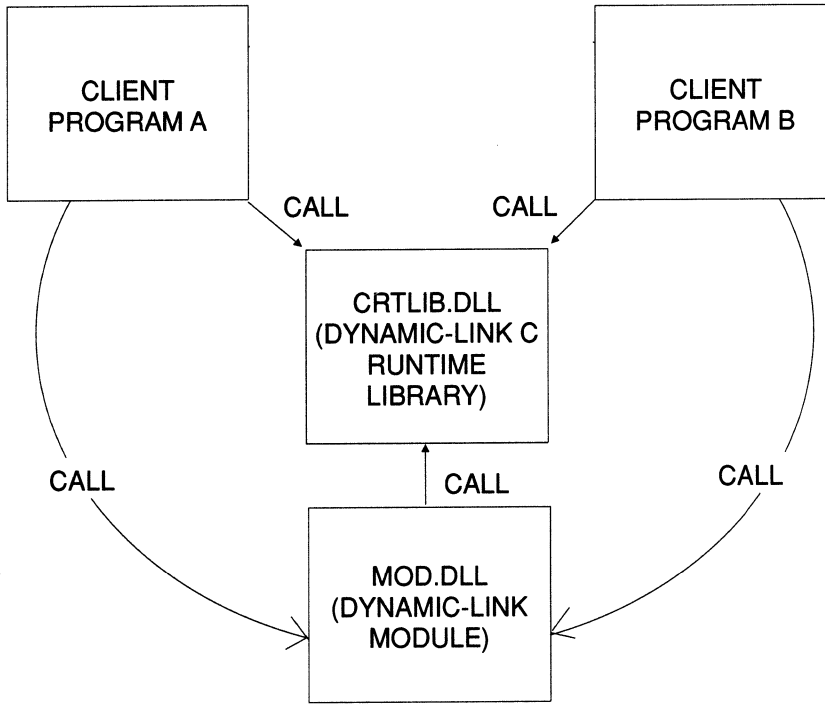
Figure 7-11

```
#    Figure 7-11
#    A MAKE file for preparing the dynamic-link module of Figure 7-8, and the
#    client program of Figure 7-10.
#
FIG7_8.OBJ : FIG7_8.C
     cl /c /W2 /ALw /G2s /Zp FIG7_8.C

FIG7_8.DLL : FIG7_8.OBJ FIG7_9.DEF
     link /NOI /NOD \PMSDK\LIB\CRTDLL.OBJ FIG7_8.OBJ, FIG7_8.DLL,, CRTLIB.LIB \
     OS2.LIB, FIG7_9.DEF

FIG7_8.LIB : FIG7_9.DEF
     implib FIG7_8.LIB FIG7_9.DEF

FIG7_10.OBJ : FIG7_10.C
     cl /c /W2 /ALw /G2 /Zp FIG7_10.C

FIG7_10.EXE : FIG7_10.OBJ FIG7_8.LIB
     link /NOI /NOD \PMSDK\LIB\CRTEXE.OBJ FIG7_10.OBJ, FIG7_10.EXE, NUL.LST, \
     FIG7_8.LIB CRTLIB.LIB OS2.LIB;
```

The Microsoft C compiler does not simply provide a single dynamic-link library file containing the entire C library. Rather, it provides the C library code as a collection of object modules contained in a library file, CDLLOBJS.LIB. You can use these object modules to create a custom

dynamic-link library incorporating only the C functions required for your dynamic-link module and client programs (or group of modules and client programs). The compiler supplies complete, detailed instructions for preparing such a custom dynamic-link C library file, as well as a batch file (MKMTDLL.CMD) for automating the preparation. When you have completed building this library, you will have both a dynamic-link library file (CRTLIB.LIB) and a corresponding import library (CRTLIB.LIB); these two files are then used when creating programs and dynamic-link modules.

Figure 7-12: Calling relationships among a dynamic-link module.



As you can see in the MAKE file of Figure 7-11, both the dynamic-link module and the client program are compiled with the usual set of flags, specifying the custom memory model /ALw (the need for this model is explained later in the section). Since both the dynamic-link module and the client program are dynamically linked to the CRTLIB version of the C library, you must specify the import library CRTLIB.LIB when linking these files.

You also need to statically link your code with an appropriate C startup routine. Accordingly, you must not define the variable *_acrtused* within the dynamic-link module source file. Note that the CRTDLL library supplies separate versions of the startup routine for programs and for dynamic-link modules. These versions are contained in distinct object modules. The startup routine for programs is contained in the file CRTEXE.OBJ, and receives initial control when a program is run. The startup routine for dynamic-link modules is contained in CRTDLL.OBJ, and receives control as an initialization routine. As you can see in the MAKE file of Figure 7-11, you must specify the appropriate object module when linking the dynamic-link library and when linking the program. Note that the module definition file of Figure 7-9 causes the startup code within the dynamic-link module to be executed only for the first client process (by leaving blank the initialization field of the LIBRARY statement).

The C library module, CRTLIB.DLL, also contains its own startup routine. Thus, the CRTLIB version of the C library employs three separate startup routines! Furthermore, because the C startup routine must be installed as the initialization routine for your module, you cannot install your own initialization routine.

Because the CRTLIB C library is dynamically linked, the dynamic-link module and all its client programs share a single copy of a dynamic-link function such as printf. In contrast, when using the LLIBCDLL library (for a dynamic-link module) and the standard C library (for a client program), the dynamic-link module and each of its client programs must store a separate copy of the function on disk, and load a separate copy into memory.

Finally, remember that when running the client program, both your dynamic-link module and the C runtime dynamic-link library

(CRTLIB.DLL) must be in a directory specified by the LIBPATH configuration command.

A third significant feature of the CRTLIB runtime library is that it can be used by multiple-thread programs or dynamic-link libraries. Remember, however, that like the multiple-thread LLIBCMT library described in the first section of the chapter, you must observe the following guidelines:

- You must manage additional threads using the functions _beginthread and _endthread rather than the OS/2 functions DosCreateThread and DosExit.
- If a program uses multiple threads, you must compile it with the /Aw flag (as you have seen, dynamic-link modules are always compiled with this flag or the /Au flag).
- You must use the special multiple-thread header files (normally contained in the MT subdirectory) for both programs and dynamic-link modules. Before including any of these header files, you must define the symbolic constant DLL, as in the following line:

```
#define DLL
```

Defining this constant causes the compiler to include the appropriate header information for dynamic-link libraries. If you are including these header files in a program that uses the other multiple-thread C library version, LLIBCMT, you must not define DLL.

A final feature of the CRTLIB library is that only a large memory model version is supplied. Since programs using this library include the special multiple-thread header files (containing explicit far declarations), you can compile your programs and modules using the small, medium, or compact memory models. See the section on the LLIBCMT library (in the first section of the chapter) for a description of the possible pitfalls that can occur when not compiling your code under the large (or huge) model. Also, remember that you must disable stack checking if compiling a program under the small or compact memory model. Stack checking should always be disabled for a dynamic-link module.

Conclusion

Table 7-1 summarizes the basic features of the standard C runtime library and the three special library versions discussed in this chapter.

Table 7-1: Basic Features of C Runtime Library Versions

LIBRARY VERSION	CODE THAT CAN USE LIBRARY	LINKING METHOD	PROGRAM THREADS SUPPORTED	MEMORY MODELS SUPPORTED	HEADER FILES
Standard	programs	static	single	small medium compact large huge	standard
LLIBCMT	programs	static	multiple	large	MT version (don't define DLL)
LLIBCDLL	dynamic-link modules	static	simgle	large	standard
CRTLIB	programs and dynamic-link modules	dynamic	multiple	large	MT version (define DLL)

Table 7-1 indicates that the CRTLIB version of the C library provides the best general resource for developing dynamic-link modules. Remember, however, that if you write a dynamic-link module using this version, all client programs that call your module must also employ the same C library version. This limitation may be unacceptable for developing general-purpose dynamic-link modules. The user of a general-purpose dynamic-link module should be able to call the functions it contains from a wide variety of program environments—even from programs written in

languages other than C. A general-purpose dynamic-link module should provide an extension to the basic OS/2 API, and the API functions certainly do not constrain the type of runtime library that may be used by the client program! Also, if you use the other library version that supports dynamic-link modules (LLIBCDLL), client programs are limited to those that run a single-program thread. Otherwise you must use semaphores to explicitly serialize calls to the C functions within the dynamic-link code.

Consequently, given the versions of the C runtime library that are currently available, it is best to develop general-purpose dynamic-link modules without using the C library functions. Modules that do not call C functions are also free to provide their own initialization routines, since the initialization privilege is no longer appropriated by the C startup code. You might want initially to develop a general-purpose dynamic-link library using the C library as a convenience, and then gradually replace the C functions with custom routines as your module approaches its final version.

CHAPTER 8

USING RUNTIME DYNAMIC LINKING

This chapter is not about creating dynamic-link libraries, but rather about an alternative method for using dynamic-link libraries within application programs.

All the example client programs that have been given so far have used **loadtime** dynamic linking. As you have seen, under this method the linking process is largely invisible to the client process. The program simply calls a dynamic-link function as an external function; when the program is executed, the system automatically loads the dynamic-link library and supplies the function address.

Alternatively, when using **runtime** dynamic linking, the client program explicitly calls the system to load the dynamic-link library and to obtain the address of the desired dynamic-link function. This process can be performed at any time during the execution of the program.

In general, a given dynamic-link library can be linked using either loadtime or runtime dynamic linking. Although runtime dynamic linking is more complex than loadtime dynamic linking, it is also a more flexible mechanism, which can be used to save loading time and to reduce the demand for memory.

This chapter first describes the basic steps required to perform runtime dynamic linking, and then discusses the advantages of this method. Next

there is a short technical digression on the topic of the **disjoint descriptor space**; this section concludes the discussion on virtual memory begun in Chapter 5. Finally, an example application illustrates several convenient methods for efficiently linking a program to a dynamic-link library at runtime.

The Basic Steps

Runtime dynamic linking is performed through the following four basic steps:

1. Call **DosLoadModule** to obtain a handle to the dynamic-link module.
2. Call **DosGetProcAddr** to obtain the address of each dynamic-link function you want to call.
3. Call the dynamic-link functions using the addresses obtained in step 2.
4. Call **DosFreeModule** to release the module when the program has completed using it.

Step 1

The first step is to call DosLoadModule to obtain a handle to the dynamic-link module containing the desired functions. DosLoadModule is described in Figure 8-1.

Figure 8-1: The DosLoadModule OS/2 function.

DosLoadModule

☐ Purpose: Loads a dynamic-link library into memory (if it has not already been loaded) and supplies a handle to this module.

☐ Prototype: USHORT APIENTRY DosLoadModule

PSZ pszFailName,	Address of a buffer that receives the name of the dynamic-link module (if any) that caused the function to fail.

USHORT cbFileName,	Length of the buffer pointed to by pszFail-Name.
PSZ pszModName,	String containing the simple filename of the dynamic-link library; the name may not include the file extension (.DLL), nor may it specify a drive or directory path.
PHMODULE phmod);	Address of the variable to receive the module handle.

☐ Return Value: If successful, the function returns zero. If an error occurs, it returns a nonzero error code. The following are the possible error codes:

ERROR_BAD_FORMAT

ERROR_FILE_NOT_FOUND

ERROR_INTERRUPT

ERROR_NOT_ENOUGH_MEMORY

The following call to DosLoadModule obtains a handle to the dynamic module that is defined in Figure 4-2:

```
USHORT Error;
char FailName [13];
HMODULE ModuleHandle;
.
.
.
Error = DosLoadModule
     FailName,
            sizeof (FailName),
            "FIG4_2",
            &ModuleHandle);
if (Error)
```

```
{
fprintf (stderr, "Error loading %s\n", FailName);
exit (1);
}
```

This function call loads the dynamic-link library FIG4_2.DLL into memory—if necessary—and assigns a handle for this module to the variable *ModuleHandle*. If the library has already been loaded for another process, DosLoadModule loads only the instance data segments. The handle will be used to refer to the module when making subsequent system calls. If DosLoadModule cannot find the specified library file, it returns a nonzero error code and writes the name of the missing module to the buffer **FailName**. The name written to FailName is not necessarily the name of the module specified in the call to DosLoadModule. If the specified module references other modules, and if one of the other modules cannot be found, the name of the missing module is written to FailName.

Obviously, when you call DosLoadModule you must know which dynamic-link library file contains the desired function (or functions). Remember from Chapter 4 that you can obtain a list of the names of the functions defined in a particular dynamic-link library by using a utility such as EXEHDR.EXE (from Microsoft). Accordingly, if you want to link with an OS/2 API function at runtime, you could examine the dynamic-link library files supplied with the system and determine which file contains the desired function (you would discover, for example, that the file VIOCALLS.DLL contains the Vio API functions).

The dynamic-link module's initialization routine (if any) is executed before DosLoadModule returns control to the client program. As mentioned in Chapter 6, if the initialization routine returns a value of zero to indicate that it has failed, the system quietly terminates the program, and DosLoadModule never returns.

Step 2

Once you obtain a handle to the dynamic-link module, the next step is to call DosGetProcAddr to get the address of each function within this module that you want to call. DosGetProcAddr is described in Figure 8-2.

Figure 8-2: The DosGetProcAddr OS/2 function.

DosGetProcAddr

☐ Purpose: Obtains the address of a dynamic-link function in memory.

☐ Prototype: USHORT APIENTRY DosGetProcAddr

(HMODULE hmod,	Handle of the module containing the function you want to call; you can obtain a handle by calling DosLoadModule (Figure 8-1).
PSZ pszProcName,	Address of a string containing either the entry point name of the function or the ordinal value of the function. If the string supplies the ordinal value, it must begin with the '#' character followed by the ASCII digits for the ordinal value. Alternatively, you can specify an ordinal value by passing an address with a selector value of zero and an offset value equal to the ordinal number.
PPFN ppfnProcAddress);	Address of the pointer variable to receive the function address.

☐ Return Value: If successful, the function returns zero. If an error occurs, it returns a nonzero error code. The following are the possible error codes:

ERROR_INTERRUPT

ERROR_INVALID_HANDLE

ERROR_PROC_NOT_FOUND

For example, the following code fragment obtains the address of the function **PrtPosition**, which is contained in the dynamic-link module loaded in the previous example (FIG4_2.DLL):

```
unsigned (pascal far *PPrtPosition) (char far *, int, int);
.
.
```

```
DosGetProcAddr
      (ModuleHandle,
      "PRTPOSITION",
      &PPrtPosition);
```

A call to DosGetProcAddr gets the address of only a single function. The example program given in the last section of the chapter shows an efficient method for obtaining the addresses of an entire module of functions.

The first parameter passed to DosGetProcAddr is the module handle previously received from DosLoadModule. The second parameter specifies the function entry name; because PrtPosition was declared as a pascal function, you must write its name in all capital letters. You can also specify the entry point by giving the ordinal value of the function; this can be done in one of two ways. First, you can pass a string containing a '#' character followed by the ordinal number, as in the following function call (like the call in the previous example, it obtains the address of PrtPosition; recall that PrtPosition was assigned an ordinal value of 5 in the module definition file of Figure 4-4):

```
DosGetProcAddr
      (ModuleHandle,
      "#5",
      &PPrtPosition);
```

Second, you can specify the ordinal value by passing as the second parameter a pointer that has a zero selector value and an offset value equal to the ordinal number of the function. The zero selector value notifies the system that the pointer does not contain a valid address pointing to a string, but rather contains the ordinal value in the offset portion of the address. The following function call is equivalent to the previous call:

```
DosGetProcAddr
      (ModuleHandle,
```

```
MAKEP (0, 5),
&PPrtPosition);
```

MAKEP is a macro defined in the OS/2 header files that creates a far pointer from a selector value (the first argument) and an offset value (the second argument). This method for specifying the ordinal entry point is illustrated by the example program given in the last section of the chapter.

You must also declare a function pointer and pass its address as the third parameter; the system assigns the address of the function in memory to this pointer, and you can use the pointer to call the function as described in the next section. You could declare a simple generic pointer, such as in the following statement:

```
unsigned (pascal far *ProcAddr) ();
```

Such a function pointer could be used for receiving the address of—and for calling—any of the functions within the printer module (FIG4_2.DLL). The example given at the beginning of this section, however, declares a function pointer (**PPrtPosition**) that specifies all of the parameter types, and is thus suitable for calling only the function PrtPosition. No other function in the module has these same parameters. If you want to call another function in the module, you have to declare another suitable pointer. The following function pointer could be used to call **PrtPutS**:

```
unsigned (pascal far *PPrtPutS) (char far *);
```

In the next section, you will see the advantage of declaring a special pointer for each function, which specifies the types of all parameters.

Step 3

Once you have received the address of a dynamic-link function in a function pointer, you can use this pointer to call the function. For example, the following statement calls PrtPosition:

```
(*PPrtPosition) ("Report Title", 1, 1);
```

Since the pointer PPrtPosition was declared specifying the type of each parameter, the compiler automatically checks the number and types of the parameters in the function call. It also converts—if necessary —the string address passed as the first parameter to the required far address.

In contrast, if you call a function using a simple generic pointer (one that does not specify the parameter types), the compiler will not check the number and types of the parameters, and it will not automatically perform required type conversions. For example, if you received the address of PrtPosition in the generic function pointer **ProcAddr**, declared as shown in the last section, you might attempt to call this function as follows:

```
(*ProcAddr) ("Report Title", 1, 1);
```

However, since the compiler does not know the parameter types for the function that is being called, it will not convert the string address passed as the first parameter to a far pointer. This conversion is required for small and medium memory model programs; accordingly, for these programs, this function call would generate a protection fault. If you employ a generic function pointer, you can overcome this problem by using appropriate cast operations as described in the last section of the chapter.

Step 4

When the program has completed using a module of dynamic-link functions, it can call **DosFreeModule** to release the module. This function is described in Figure 8-3. DosFreeModule does not actually release the module from memory until it has been freed by the last client process. Although the module may still be resident in memory, after calling DosFreeModule the handle supplied by DosLoadModule can no longer be used, and the program can no longer call functions within the module. Calling a function would cause a protection fault.

Calling DosFreeModule is optional; when the process terminates, the system automatically releases the module. However, if the program has finished using a particular module before it terminates, calling DosFreeModule may reduce memory needs during the remainder of the execution of the program.

Figure 8-3: The DosFreeModule OS/2 function.

DosFreeModule

☐ Purpose: Releases a module previously loaded by DosLoadModule (Figure 8-1).

☐ Prototype: USHORT APIENTRY DosFreeModule

(HMODULE hmod); Module handle supplied by DosLoadModule.

☐ Return Value: If successful, the function returns zero. If an error occurs, it returns a nonzero error code. The following are the possible error codes:

ERROR_INTERRUPT

ERROR_INVALID_HANDLE

Advantages of Runtime Dynamic Linking

As you can see, using runtime dynamic linking is considerably more complex than using loadtime dynamic linking. However, runtime dynamic linking is a more flexible method and offers several advantages important for certain types of applications.

First, under loadtime dynamic linking any module referenced by the program is invariably loaded when the program is first run. Under runtime dynamic linking, however, the program does not need to load a module unless it is actually used. For instance, the example program presented in the last section of this chapter loads the module of printer functions only if the user chooses to print a report.

Second, when using runtime dynamic linking, the loading of a module is postponed until the functions in the module are first required; in the meantime, memory is conserved. Also, to save memory the program can load and release one module at a time, so that only the module or modules currently in use need to be stored in memory. This advantage would *not* apply if the module is held in memory by another client program.

Third, under loadtime dynamic linking the name of the module is hard coded into the executable file header. Under runtime dynamic linking,

however, the program can load one of several alternative modules depending upon current requirements. For example, you might develop separate versions of a library of video routines for different display systems; the program would load the appropriate version depending upon the current video configuration.

Finally, under loadtime dynamic linking the system automatically aborts the process if it cannot find a referenced dynamic link library, displaying the following message:

```
SYS1804 The system cannot find the file xxxx
```

In this message *xxxx* is the name of the missing module. Under runtime dynamic linking, however, if the system cannot find a dynamic-link library, DosLoadModule simply returns an error code and supplies the name of the missing module. The program can then take appropriate action, such as notifying the user that a particular set of functions is unavailable, or attempting to load an alternative module.

The Disjoint Descriptor Space

This section presents a short technical digression into the topic of the disjoint descriptor space, and completes the discussion on virtual memory begun in Chapter 5. This subject is presented to enhance your theoretical understanding of the dynamic-link mechanism, and is not required for comprehending other material in the book.

When a program calls DosLoadModule, the dynamic-link module is loaded using the same method employed under loadtime dynamic linking. Specifically, if the module has not already been loaded for another process, the system loads all segments defined by the module and grants the current process access to these segments. If, however, the module has already been loaded, the system reloads only the instance data segments (if any), and the current process shares all code and global data segments already in memory. A dynamic-link code segment typically contains segment addresses referring to code or data segments belonging to the module; for example, in the machine instruction:

```
MOV AX, DGROUP
```

the symbol DGROUP refers to the segment address of the automatic data segment. When a code segment is loaded for the first client process, the system writes the appropriate segment selector values to all segment address references within the code. These selector values are not known until runtime; the process of filling in the actual segment selector values is known as **relocation.**

Once the segment values have been written to a code segment, these values are never altered (in other words, relocation is performed only when the code segment is loaded for the first client process). Accordingly, when the second (or subsequent) client process begins sharing the code segment, it sees the same selector values. As explained in Chapter 5 (in the Virtual Memory section), these segment selectors serve as indices into the local descriptor table (LDT) belonging to the current process. When the second client process is executing a dynamic-link function and it encounters a segment selector within the function code, the corresponding entry in its local descriptor table must point to the appropriate segment in memory. This segment can be global or instance; see Figure 5-9.

Therefore, when the second process calls DosLoadModule, the system must assign the correct physical addresses to all of the process's descriptor table entries that are indexed by segment selectors already contained in the dynamic-link code.

At this point you may see a potential problem. What if one or more of these LDT entries has already been used to contain the descriptor for a segment previously allocated to the second process? By the time a process calls DosLoadModule it has already been allocated all of the segments defined in the EXE file, as well as any segments obtained dynamically from the system. Consequently, many of the slots in its local descriptor table may be occupied. The answer to this question requires a short explanation of the disjoint descriptor space.

The local descriptor table belonging to each process has a large number of potential entries. This discussion refers to these potential entires as **slots**; a slot can hold a single segment descriptor. Under the 80286 processor, each LDT has 8,192 slots. OS/2 version 1.0 reserved approximately half of these slots for holding descriptors for the segments defined by dynamic-link libraries; these reserved slots constitute the disjoint descriptor space. The system also uses the disjoint descriptor

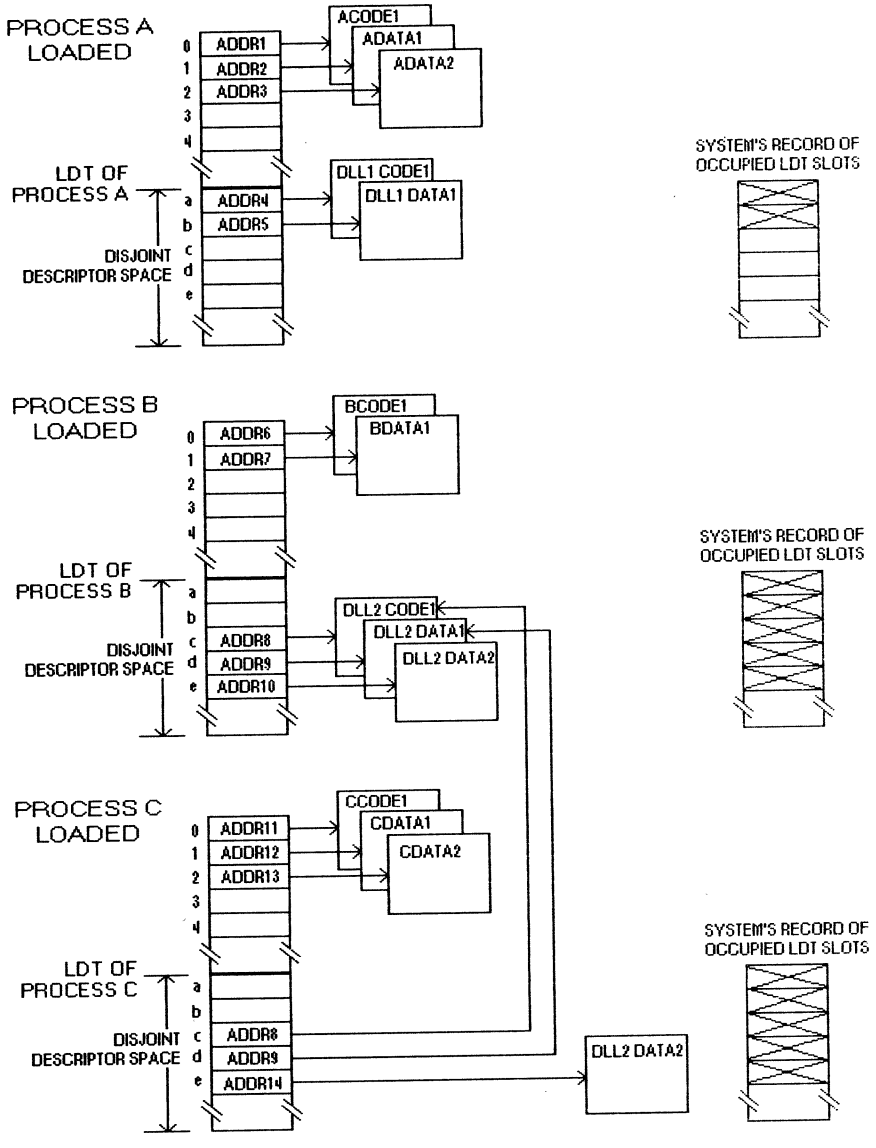
space for shared segments in general; the allocation of shared segments, however, is beyond the scope of this discussion.

Figure 8-4 illustrates how the disjoint descriptor space is used to assure that the required LDT slots are available when a dynamic-link library is loaded at runtime. Figure 8-4 depicts the loading of three processes: first Process A, then Process B, and finally Process C. Process A defines three segments: one code segment (ACODE1) and two data segments (ADATA1 and ADATA2). When the system loads Process A, it places the descriptors for these segments within normal, non-disjoint slots of Process A's LDT (slots 0, 1, and 2). Process A also references a dynamic-link library (DLL1), which defines one code segment (DLL1 CODE1) and one data segment (DLL1 DATA); the descriptors for these segments are placed within the disjoint descriptor space portion of Process A's LDT (slots *a* and *b*). The system then marks these two slots as **occupied** (in the diagram, occupied slots are marked with an *X* in the system's record of occupied LDT slots). Accordingly, no subsequently loaded process can use the corresponding slots within its own LDT, except for referring to the same dynamic-link segments.

The system then loads Process B, which defines two segments: the system places the descriptors for these segments in slots 0 and 1 of Process B's LDT. Process B also references a dynamic-link module—DLL2—a different dynamic-link library than that used by Process A. This module defines three segments. As before, the system places the descriptors for these segments within the disjoint descriptor space.

However, since slots *a* and *b* are marked as occupied, the system uses three other slots (*c*, *d*, and *e*), and then marks these additional slots as occupied. At this point, slots *a* through *e* are all marked as occupied. Remember that when the system loads the code belonging to the dynamic-link library DLL2, it writes the appropriate selector value (*c*, *d*, or *e*) to all segment address references within this code—it performs relocation, since the code is being loaded for the first client process.

Figure 8-4: The use of the disjoint descriptor space.



The system finally loads Process C, and as before it places the descriptors for the segments defined by this process within the non-disjoint portion of its LDT. Process C also references the same dynamic-link library as Process B (DLL2). Since Process C shares the dynamic-link code with Process B, and since the system has already performed address relocation on this code, when Process C executes the dynamic-link code, it may encounter the segment selector values *c*, *d*, or *e*. Consequently, slots *c*, *d*, and *e* within its own LDT must refer to the appropriate dynamic-link segments. Fortunately, the system reserved these slots; therefore, it can now place the appropriate segment descriptors within these slots. You can see from Figure 8-4 that the segments named "DLL2 CODE1" and "DLL2 DATA1" are global segments, and the segment named "DLL2 DATA2" is an instance segment.

An Example Application

Figure 8-5 lists a program that uses runtime dynamic linking to access the module of printer functions defined in Figure 4-2. This program serves to demonstrate the following techniques:

- Converting a program that uses loadtime dynamic linking to one that uses runtime dynamic linking.
- Obtaining the addresses of an entire module of functions using a simple program loop.
- Using **#define** statements to simplify calling functions dynamically linked at runtime.

The example program first displays a menu giving the user the choice of printing a report or terminating the application. If the user chooses to run the report, the program uses the module of dynamic-link printer functions to print a simulated report. These are the same basic tasks performed by the example program of Figure 8-5, which accesses the printer module using loadtime dynamic linking. Note also that the two functions in the program of Figure 8-5 that use the dynamic-link printer functions (**PrintReport** and **Header**) are identical to the two functions that perform the same tasks within the program of Figure 4-7.

Figure 8-5

```
/*
     Figure 8-5

     This program uses runtime dynamic linking to load and call the following
     functions defined in the dynamic-link library of Figure 4-2:

          PrtReady
          PrtInit
          PrtPutS
          PrtNewPage
          PrtPosition

     The program can be built using the following commands:

          cl /c /W2 /G2 /Zp FIG8_5.C
          link /NOI /NOD FIG8_5.OBJ,, NUL, SLIBCE.LIB OS2.LIB;
*/
#define INCL_DOS
#include <OS2.H>

#include <STDIO.H>
#include <CONIO.H>
#include <PROCESS.H>

/* The following declaration and definitions are used for runtime dynamic    */
/* linking and replace the module header file of Figure 4-3.                 */

void (pascal far *PointerTable [7]) ();       /* Holds the addresses of all  */
```

```
                                        /* dynamic-link functions.      */
                    /* The following definitions allow the dynamic-link      */
                    /* functions linked at runtime to be called using the    */
                    /* SAME calling protocol as functions linked at loadtime:*/

#define PrtReady    (*((unsigned (pascal far *) \
                         (unsigned char far *)) PointerTable [1]))

#define PrtInit     (*((unsigned (pascal far *) \
                         (void)) PointerTable [2]))

#define PrtPutC     (*((unsigned (pascal far *) \
                         (int)) PointerTable [3]))

#define PrtPutS     (*((unsigned (pascal far *) \
                         (char far *)) PointerTable [4]))

#define PrtPosition (*((unsigned (pascal far *) \
                         (char far *, int, int)) PointerTable [5]))

#define PrtNewPage  (*((unsigned (pascal far *) \
                         (unsigned)) PointerTable [6]))

/*  The following are the local functions and external variables used in     */
/*  the program:                                                             */

void LoadModule (void);         /* Loads dynamic-link module.               */
void PrintReport (void);        /* Uses dynamic-link functions to print report.*/


HMODULE ModuleHandle;           /* Handle to dynamic-link module.           */

void main (void)
```

```
    {
    int Choice;

    printf ("Programs Options:\n");              /* Display a menu.          */
    printf ("     (1)  Print Report\n");
    printf ("     (2)  Terminate Program\n");
    printf ("Select 1 or 2:  ");

    for (;;)
         switch (getch () - '0')
              {
              case 1:
                   LoadModule ();                /* Load functions.         */
                   PrintReport ();               /* Use the functions.      */
                   DosFreeModule (ModuleHandle); /* Release the module.     */

              case 2:
                   exit (0);
              }

    } /* end main */


/* LoadModule loads the dynamic-link module and obtains the addresses of the */
/* dynamic-link functions:                                                   */

void LoadModule (void)
     {
     USHORT Error;            /* Receives API error code.                    */
     char FailName [13];      /* Used by 'DosLoadModule'.                    */
```

```
    int Ordinal;              /* Ordinal values of dynamic-link functions.   */
    Error = DosLoadModule     /* Load the dynamic-link module.               */
         (FailName,           /* Receives name of file causing failure.      */
          sizeof (FailName),  /* Length of 'FailName' buffer.                */
          "FIG4_2",           /* Name of dynamic-link module.                */
          &ModuleHandle);     /* Receives handle to dynamic-link module.     */
    if (Error)
         {
         printf ("Error loading %s\n", FailName);
         exit (1);
         }

    /* Obtain the address of each dynamic-link function and assign it to    */
    /* 'PointerTable'.                                                      */

    for (Ordinal = 1; Ordinal <=6>; ++Ordinal)
         DosGetProcAddr
              (ModuleHandle,                /* Dynamic-link module handle.  */
              MAKEP (0, Ordinal),           /* Function ordinal value.      */
              &PointerTable [Ordinal]);     /* Receives function address.   */

    } /* end LoadModule */

/* Report printing data structures and functions:                           */

static void Header (void);              /* Prints report headers.           */
static int Row;                         /* NEXT row to be printed.          */

void PrintReport (void)
    {
```

```
unsigned ErrorCode;                 /* Saves error code.                */
int i;                              /* Loop index.                      */
unsigned char FlagReady;            /* Flag indicating printer ready.   */

                                  /* Make sure that printer is ready:   */
while (!(ErrorCode = PrtReady (&FlagReady)) && !FlagReady)
     {
     printf ("\nReady printer and press any key to continue ...");
     getch ();
     }
if (ErrorCode)
     {
     printf ("PrtReady error %d\n",ErrorCode);
     exit (1);
     }

printf ("\nPrinting report...");

ErrorCode = PrtInit ();             /* Initialize printer.              */
                                    /* Number of selected printer.      */

                                    /* Send control code sequence for near */
ErrorCode = PrtPutS                 /* letter quality (Okidata).        */
     ("\x1b\x49\x33");              /* Control code string.             */
if (ErrorCode)
     {
     printf ("PrtPutS error %d\n",ErrorCode);
     exit (1);
     }

ErrorCode = PrtNewPage              /* Initialize a new page without formfeed.*/
```

```
        (0);                        /* Flag indicates NO formfeed.          */
    if (ErrorCode)
        {
        printf ("PrtNewPage error %d\n",ErrorCode);
        exit (1);
        }
    Header ();                      /* Print first header.                  */
    for (i = 1; i <= 80; ++i)       /* Process 80 detail lines.             */
        {
        if (Row >  55)
            {
            PrtNewPage (1);                 /* New page with a formfeed.    */
            Header ();                      /* Print another header.        */
            }
        PrtPosition ("Field One",Row,1);
        PrtPosition ("Field Two",Row,23);
        PrtPosition ("Field Three",Row,44);
        PrtPosition ("Field Four",Row++,67);
        }
    PrtPosition ("End of Report",++Row,1);
    PrtNewPage (1);                 /* Force out last page.                 */
    } /* end PrintReport */

static void Header (void)
    {
    PrtPosition ("S A M P L E   R E P O R T",1,27);
```

```
PrtPosition ("Heading One",3,1);
PrtPosition ("Heading Two",3,23);
PrtPosition ("Heading Three",3,44);
PrtPosition ("Heading Four",3,67);
PrtPosition ("-----------",4,1);
PrtPosition ("-----------",4,23);
PrtPosition ("-------------",4,44);
PrtPosition ("------------",4,67);
Row = 6;

} /* end Header */
```

The program in Figure 8-5 was created by modifying the program of Figure 4-7 in such a way that it could use runtime dynamic linking without changing the calling protocol for the dynamic-link functions. Accordingly, the main body of code (contained in the functions PrintReport and Header) is identical in both programs, simplifying the conversion process. As you can confirm by comparing Figures 4-7 and 8-5, you can convert a program that uses loadtime dynamic linking to one that uses runtime dynamic-linking through the following three steps:

1. Replace the module header file (which contains declarations for the dynamic-link functions and is listed in Figure 4-3 for the example program) with: (1) the definition of an array of function pointers (**PointerTable**), and (2) a #define statement for each function, which allows you to call the function the using the conventional calling protocol.

2. Before using the dynamic-link functions, call a function (LoadModule) that loads the dynamic-link module and places the address of each function within the array of function pointers.

3. After using the dynamic-link functions, call DosFreeModule to release the module from memory (this step is optional; the dynamic-link library will automatically be released when the process terminates, unless another process is still using it).

Even if you are not converting an existing program that uses loadtime linking, the techniques described in this section make it simpler to load and call dynamic-link functions at runtime. Since you are using runtime dynamic linking, when you link the program, you no longer need to specify the module import library. You can use the following two commands to prepare the program of Figure 8-5:

```
cl /c /W2 /G2 /Zp FIG8.C
```

and:

```
link /NOI /NOD FIG8.OBJ,, NUL, SLIBCE.LIB OS2.LIB;
```

Figure 8-5's program defines an array of function pointers, as follows:

```
void (pascal far *PointerTable [7]) ();
```

This array is first used to receive the addresses of the functions belonging to the dynamic-link module, and it is then used to call these functions. The advantage of storing the dynamic-link function addresses in an array is that the program can obtain the addresses for an entire set of functions by calling DosGetProcAddr within a simple loop—see the function LoadModule, in Figure 8-5. When calling DosGetProcAddr, the functions are referenced by their ordinal entry point values; remember that the functions in this module were assigned ordinal values 1 through 6 (in the definition file of Figure 4-4). The program uses only positions 1 through 6 in this array so that the index of an array entry will be the same as the ordinal value of the function whose address is stored in this entry.

The base type of the array PointerTable is a generic function pointer that specifies neither the function return type nor its parameters. If you were to call a function directly through a member of this array, you would have to employ an expression such as the following:

```
(*PointerTable [5]) ("Report Title", 1, 1);
```

This expression calls the function PrtPosition, which was assigned an ordinal value of 5. However, there are two serious problems with such a function call. First, any code already developed for loadtime dynamic

linking would have to be rewritten, replacing simple function calls with awkward indirection operations on function pointers. Second, the compiler is unable to check the type and number of parameters, and will not perform necessary parameter type conversions. The function call listed here would cause a protection fault within a small or medium memory model program. The solution to these two problems is to provide a #define statement for each dynamic-link function, such as the following:

```
#define PrtPosition (*((unsigned (pascal far *) \
                    (char far *, int, int)) PointerTable [5]))
```

This definition not only allows you to call the function directly by name, but also casts the generic function pointer to the exact type for the specific function, specifying all parameters. The program can subsequently call the function exactly as if it were linked at loadtime, and the compiler will check the parameter types and perform all required type conversions. For example, the function PrtPosition could be called as follows:

```
PrtPosition ("Report Title", 1, 1);
```

The program of Figure 8-5 illustrates an important advantage of runtime dynamic linking: it does not load the dynamic-link module unless the functions in this module are required. The module is loaded only if the user chooses the "Print Report" menu item. In contrast, the program of Figure 4-7 always loads the module when the program is first run, regardless of whether the module is ever used. A more complex application could use runtime dynamic linking to load one of several alternative modules, depending upon the program's needs. For example, there might be a separate dynamic-link module for managing each type of printer supported by the application; once the program determines the type of printer installed, it could load the required module version.

CHAPTER 9

REAL-MODE VERSION OF YOUR LIBRARY

All of the example programs you have seen so far in this book can be run only in a protected-mode environment—either from the OS/2 command prompt or within the Presentation Manager. If you attempt to run one of these programs under MS-DOS, or within the DOS compatibility box of OS/2, the system immediately prints an error message and terminates the loading process.

Under OS/2, however, you can develop a program with a special format that allows it to be run under *either* protected mode *or* real mode. These programs are termed **dual-mode** or **bound** applications, and can be run within three distinct environments: MS-DOS, the OS/2 DOS compatibility box, and an OS/2 protected-mode screen group. You may be able to run a dual-mode program within a window of the Presentation Manager; however, for reasons that will soon become obvious, a full Presentation Manager application cannot be converted to the dual-mode format.

Many applications can be made into dual-mode programs since the code generated by the C compiler can generally be executed in either real or protected mode. Most OS/2 programs, however, contain calls to dynamic-link libraries, and dynamic linking is not supported within real mode. Accordingly, to permit the creation of dual-mode programs, OS/2 provides

special real-mode versions of most of the API functions. Through a process that will be explained in this chapter, the real-mode routines can be substituted for the dynamically linked, protected-mode versions when the program is loaded under real mode.

You can also provide real-mode versions of the dynamic-link libraries that you develop, so that you or other users of your library routines can call these functions within dual-mode programs.

The first section of the chapter describes how to create dual-mode programs. The next section explains the step-by-step procedures for writing real-mode versions of your dynamic-link library functions. The final section describes how to use these real-mode functions to create a dual-mode program that calls your module.

Creating Dual-Mode Programs

The first step in creating a dual-mode program is to produce a standard protected-mode application using the techniques described in this book. Once you have generated the protected-mode executable file, you can then run the OS/2 BIND utility to convert this file to the dual-mode format.

The following is an example of a command line invoking the BIND utility:

```
bind PROG.EXE \LIB\OS2.LIB \LIB\API.LIB
```

This command causes BIND to convert the protected-mode program PROG.EXE into a dual-mode executable file having the same name. API.LIB is a conventional library file supplied with OS/2, which contains real-mode versions of most of the OS/2 API functions (note that you must supply the full path name of this file).

In the process of converting a protected-mode program to a dual-mode program, the BIND utility adds the real-mode versions of all dynamic-link functions called by the application; these functions are inserted into the executable file header. BIND also inserts the code used for loading these functions if the program is run under real mode. This code is known as the **stub loader**).

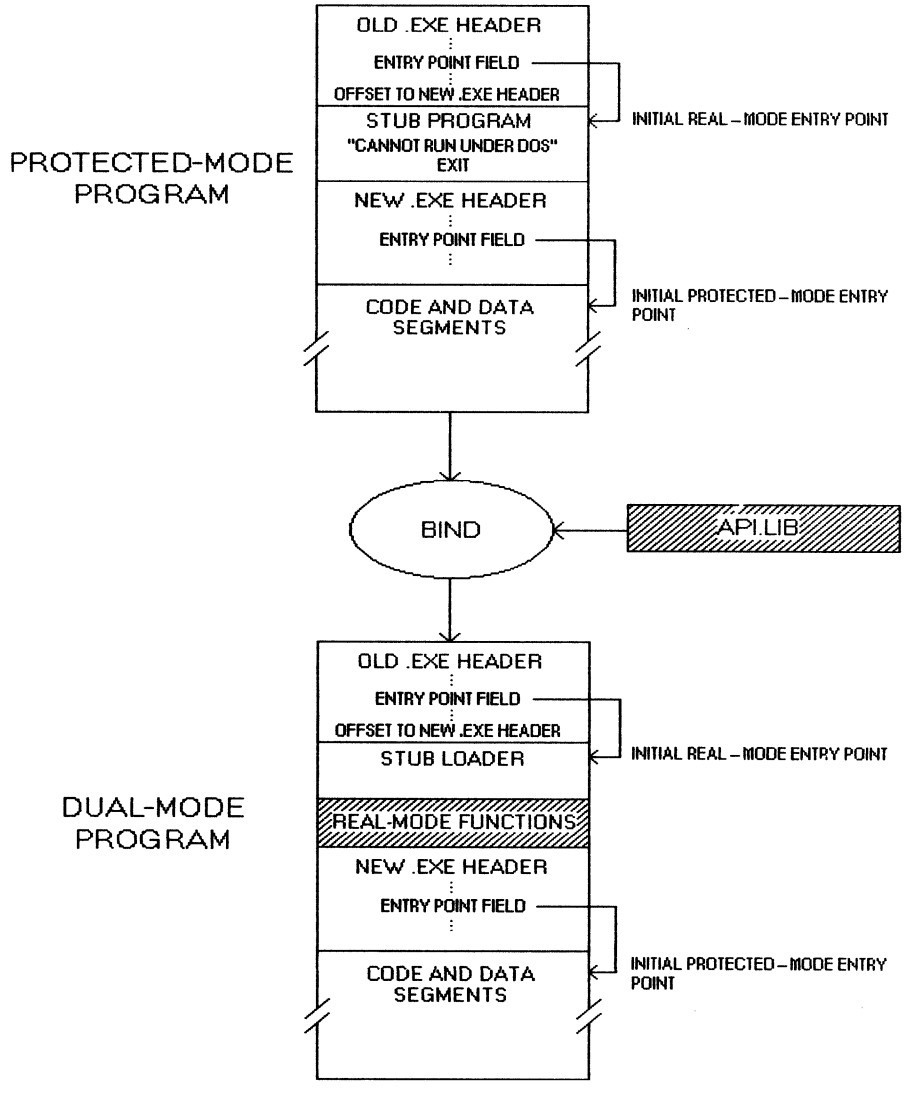
You must also specify the import library, OS2.LIB (or DOSCALLS.LIB), originally used for linking the program. The import library is needed because the conventional library containing the real-mode API function versions (API.LIB) identifies all functions by *name*. However, the dynamic-link file header (specifically, the relocation table described in Chapter 3) identifies these functions by *ordinal value*. The import library tells BIND the name of the function corresponding to each ordinal value in the file header. See the technical documentation for a full description of the syntax of the BIND command and for an explanation of its other options. Figure 9-1 illustrates a program before and after it has been converted by BIND into the dual-mode format.

Notice in this figure the similarity between a protected- mode program and a dual-mode program. Both types of programs begin with a standard MS-DOS program header (the "Old EXE Header"). The header information specific to OS/2 is found later in the file (the "New EXE Header"). Both files also have dual entry points: one that receives control under protected mode (the "Initial Protected-Mode Entry Point") and one that receives control under real mode (the "Initial Real-Mode Entry Point").

When either a protected-mode program or a dual-mode program is run under protected mode, the loader first sees a standard MS-DOS executable file header. However, a field within this header indicates that the file contains an OS/2 program—either protected mode or dual mode. Specifically, in an OS/2 program the field at offset 18h within the old header is set to 40h, a value that could never be found in a valid MS-DOS program. If this field contains some other value, the OS/2 loader prints an error message and terminates the process.

The loader next obtains the offset of the new portion of the header, which is stored immediately beyond the end of the old header. It then processes the many items of information found within the new header. Among the other tasks it performs at this time, it loads all referenced dynamic-link libraries and performs address relocation as described in Chapter 3. When the system has completed loading the program and preparing the executable image, it passes control to the address indicated in the entry point field of the new file header (the "Initial Protected-Mode Entry Point").

Figure 9-1: A program before and after conversion by BIND.



Likewise, if either a protected-mode or dual-mode program is run under real mode (that is, under MS-DOS or the DOS compatibility box of OS/2), the loader first sees a standard MS-DOS file header. This header contains, among other items of information, a pointer to the "Initial Real-Mode Entry Point." When the system has completed the loading procedure, it passes control to the entry point indicated by this field. If this were an MS-DOS program, the entry field would point to the main program entry routine; for example, in an MS-DOS C program, the C startup code would receive control. In an OS/2 program, initial control passes to a routine known as a **stub program**.

The differences between a protected-mode program and a dual-mode program lie in the actions performed by the stub program. In a protected-mode program, the stub program simply prints the following error message and terminates the futile attempt to run a protected-mode program under real mode:

```
This program cannot be run in DOS mode.
```

When BIND converts a protected-mode program to a dual-mode program, it replaces the normal stub program with a stub loader. Rather than simply terminating the program, the stub loader performs the following actions when the program is run under real mode:

- It loads the real-mode versions of all dynamic-link functions, which are contained in the file header.
- It fills in the address field of each call instruction within the program code that invokes a dynamic-link function. For each of these call instructions, rather than supplying the address of a function within a dynamic-link library, it supplies the address of the real-mode version of this function that has just been loaded into memory.
- It transfers control to the main program entry point (labeled the "Initial Protected-Mode Entry Point" in Figure 9-1).

Thus, the main program entry point ultimately receives control in either real or protected mode. In real mode, control first passes through the stub loader, and in protected mode, control passes directly to the main

entry point. Because the stub loader is bypassed in protected mode, the real-mode versions of the dynamic-link functions are never loaded; thus, in protected mode these routines consume neither loading time nor memory space.

As already mentioned, the library file API.LIB (used by the BIND utility) contains real-mode versions for most of the standard OS/2 API functions. The functions included in this library are known as the **family API subset** of the full OS/2 API. The functions included in the family API are those that can easily be emulated under MS-DOS; accordingly, this subset excludes functions that manage unique features of OS/2, such as multitasking and interprocess communication. The following groups of functions are also excluded from the family API:

- All mouse subsystem functions (Mou routines). Note that under real mode these services can be obtained through interrupt 33h.
- Functions specific to the Presentation Manager (for example, all **Gpi** and **Win** functions). Thus, you cannot convert a Presentation Manager application into a dual-mode program.

Although you can call functions in the family API from a program destined to be converted to the dual-mode format, many of these functions have limitations when called from real mode; the next section in this chapter gives an example of such a limitation. The OS/2 programmer's reference states which functions belong to the family API, and also fully documents any restrictions that apply when calling one of these functions in real mode.

However, a dual-mode program can call an OS/2 API function that is not in the family subset, provided that it conforms to two guidelines. First, the program can call the function only if it is currently running in protected mode. The OS/2 function **DosGetMachineMode** can be used to determine whether the system is running in real or protected mode.

Second, when you create the dual-mode program with the BIND utility, you must resolve the reference to the non-family function. A non-family API function will not be resolved by API.LIB, which contains only the family subset of functions. However, by listing the name of a non-family API function (or several such functions) following the /n command line flag, you can cause BIND to resolve calls to the specified function with

the address of the routine **BadDynLink.** The code for BadDynLink is inserted into the program header along with the other real-mode functions. As a result, if the program inadvertently calls the non-family API function under real mode, control passes to BadDynLink, which prints an error message and terminates the program.

As an example, the following code calls the non-family API function **DosGetInfoSeg** *only* if the program is running under protected mode:

```
BYTE ProtectedMode;
SEL GlobalSeg;
SEL LocalSeg;
.
.
.
DosGetMachineMode (&ProtectedMode)

if (ProtectedMode)
     DosGetInfoSeg
            (&GlobalSeg,
            &LocalSeg);
```

Assuming that DosGetInfoSeg is the only non-family API function in this program, you could convert the program to the dual-mode format using the following command:

```
bind PROG.EXE \LIB\OS2.LIB \LIB\API.LIB /n DOSGETINFOSEG
```

This command would resolve the call to the function DosGetInfoSeg with the address of the routine BadDynLink. Accordingly, if through a programming error the program attempted to call DosGetInfoSeg when running in real mode, the function BadDynLink would receive control and terminate the program after printing an error message, such as the following:

```
The Application Program Interface (API) entered will only work in
  IBM Operating System/2 mode.
```

If you create a dual-mode program using the BIND commands illustrated in this section, your program cannot call dynamic-link functions other than the OS/2 family API functions. To allow a dual-mode program to call the dynamic-link functions that you have developed, you must prepare a conventional library file containing real-mode versions of your functions, which would be analogous to the API.LIB library supplied with OS/2. You must then specify this library file—along with API.LIB—on the BIND command line. The remaining sections of this chapter discuss how to write and use such a library.

Writing a Real-Mode Version of a DLL

This section describes how to create real-mode versions of the functions contained in your dynamic-link library, so that these functions can be called from a dual-mode program. Figures 9-2 and 9-3 define real-mode versions of the set of printer functions originally defined in the dynamic-link module source file of Figure 4-2. Figure 9-2 is a C source file and Figure 9-3 is an assembly language file. Once you have prepared a library file containing the functions defined in Figures 9-1 and 9-2, you can write a dual-mode client program that can call any of the functions in the printer module.

Explaining the implementation details of the assembler functions in Figure 9-3 is beyond the scope of this book. However, the comments in the source file clarify many of the details; also, Chapter 10 provides further information on using assembly language.

Figure 9-2

```
/*

    Figure 9-2


    This file contains the C functions belonging to the real-mode version of

    the module of printer functions.  The assembly language functions are
```

```
     defined in Figure 9-3.  The real-mode library can be prepared using the
     MAKE file of Figure 9-4.
*/

#define INCL_DOS
#include <OS2.H>

#include "FIG4_3.H"            /* Module header file.                        */

#define PRTNAME "LPT1"         /* Printer device name.                       */

int _acrtused = 0;             /* Define variable to avoid linking in C      */
                               /* startup code.                              */

                               /* External variables for storing printer state*/
unsigned char Opened = 0;      /* Indicates whether printer has been opened. */
HFILE Handle;                  /* Handle for printer device.                 */
int CurRow = 1;                /* Current row of printer head.               */
int CurCol = 1;                /* Current column of printer head.            */

                               /* Private functions:                         */
USHORT _PrtOpen                /* Opens the printer.                         */
     (void);
USHORT _StrLen                 /* Calculates length of a string.             */
     (char far *String);

unsigned pascal far _loadds PrtPutC
     (int Ch)
/*
     This function sends character 'Ch' to the printer.  If successful, it
     returns zero;  if an error occurs, it returns a nonzero API error code.
```

```
     Warning:  Neither 'PrtPutC' nor 'PrtPutS' should not be used in conjuntion
     with 'PrtPosition' unless the function is used to send a control code that
     does NOT move the printer head (otherwise the internal record of the
     current printer row and column maintained by the module would become
     invalid).
*/
     {
     USHORT ErrorCode;              /* Stores the API error code.              */
     USHORT BytesWritten;           /* Number of bytes successfully printed.   */

     if (!Opened)                   /* Open printer if necessary.              */
          {
          ErrorCode = _PrtOpen ();
          if (ErrorCode)
               return (ErrorCode);
          }

     ErrorCode = DosWrite                    /* Send character to printer.    */
          (Handle,                           /* Printer device handle.        */
          &Ch,                               /* Address of char. to print.    */
          1,                                 /* Number of bytes to print.     */
          &BytesWritten);                    /* Assigned bytes written.       */
     if (ErrorCode)
          return (ErrorCode);

     return (0);

     } /* end PrtPutC */


unsigned pascal far _loadds PrtPutS
     (char far *String)
```

```
/*
     This function sends the NULL terminated string 'String' to the printer.
     See the warnings given for 'PrtPutC', which apply to 'PrtPutS' as well.
     If successful, it returns zero;  if an error occurs, it returns a
     nonzero API error code.
*/
     {
     USHORT ErrorCode;              /* Stores the API error code.             */
     USHORT BytesWritten;           /* Number of bytes successfully printed.  */

     if (!Opened)                   /* Open printer if necessary.             */
          {
          ErrorCode = _PrtOpen ();
          if (ErrorCode)
               return (ErrorCode);
          }

     ErrorCode = DosWrite                    /* Send string to printer.       */
          (Handle,                           /* Printer device handle.        */
          String,                            /* Address of string to print.   */
          _StrLen (String),                  /* Number of bytes to print.     */
          &BytesWritten);                    /* Assigned bytes written.       */
     if (ErrorCode)
          return (ErrorCode);

     return (0);

     } /* end PrtPutS */


unsigned pascal far _loadds PrtPosition
     (char far *String,
```

```
    int Row,
    int Col)
/*
    This function prints NULL terminated 'String' beginning at the position
    specified by 'Row' and 'Column'.  If successful, it returns zero;  if an
    error occurs, it returns one of the following error codes:

        BADPOSITION          The requested print position was to the left of
                             or above the current printer head position.

        For all other errors, it returns the API error code.

    The following rules must be observed:

    o  The string must NOT contain control characters (i.e., any characters
       that do not advance the print head a single column).  To send control
       codes, use 'PrtPutC' or 'PrtPutS'.
    o  The string must not contain tab characters.
    o  The string must not contain newline characters.  To advance to a new
       line, use a subsequent call specifying the desired row.  Do not send
       more characters than can fit on the current line.
    o  To generate a new page and reset the row and column numbers, use
       'PrtNewPage'.  Do not send more lines than can fit on a single page.
    o  'PrtPosition' and 'PrtNewPage' should be used by only a single thread
        within a process at a given time.
*/
    {
    USHORT ErrorCode;              /* Stores the API error code.            */

    /*** Test for valid row and column. ************************************/
```

```
if (Row < CurRow ||
    Row == CurRow &&
    Col < CurCol)
     return (BADPOSITION);

/*** Print CR/LF pairs until reaching desired row. **********************/
while (Row - CurRow)
     {
     ErrorCode = PrtPutS ("\x0d\x0a");
     if (ErrorCode)
          return (ErrorCode);
     ++CurRow;                  /* Adjust record of printer position.     */
     CurCol = 1;
     }

/*** Print spaces until reaching desired column. ************************/
while (Col - CurCol)
     {
     ErrorCode = PrtPutC (32);
     if (ErrorCode)
          return (ErrorCode);
     ++CurCol;                  /* Adjust record of current column.       */
     }

/*** Print the string. **************************************************/
ErrorCode = PrtPutS (String);
if (ErrorCode)
     return (ErrorCode);

CurCol += _StrLen (String);    /* Adjust record of current column.       */
```

```
    return (0);

    } /* end PrtPosition */

unsigned pascal far _loadds PrtNewPage
    (unsigned FlagFormFeed)
/*
    This function resets the internal row and column counters for the
    position of the printer head;  if 'FlagFormFeed' is nonzero, the
    function also generates a formfeed.  If successful, it returns zero;
    if an error occurs, it returns a nonzero API error code.
*/
    {
    USHORT ErrorCode;              /* Stores the API error code.          */

    if (FlagFormFeed)
        {
        /*** Generate a carriage return and form feed. *********************/
        ErrorCode = PrtPutS ("\x0d\x0c");
        if (ErrorCode)
            return (ErrorCode);
        }

    /*** Reset current row and column. ************************************/
    CurRow = CurCol = 1;

    return (0);

    } /* end PrtNewPage */
```

```
/*** Private functions: ****************************************************/

USHORT unsigned _PrtOpen
     (void)
/*
     This private function opens the printer device.  If successful, it returns
     zero;  if an error occurs, it returns a nonzero API error code.
*/
     {
     USHORT ErrorCode;              /* Stores the API error code.           */
     USHORT Action;                 /* Receives 'DosOpen' action code.      */

     ErrorCode = DosOpen
          (PRTNAME,            /* Device name for printer.                   */
          &Handle,             /* Receives printer device handle.            */
          &Action,             /* Receives action code.                      */
          0L,                  /* Initial allocation size:  n/a.             */
          0,                   /* File attribute:  n/a.                      */
          1,                   /* Open flag:  open file if it exists.        */
          0x0041,              /* Open mode:  write access and share.        */
          0L);                 /* Reserved:  must be 0.                      */
     if (ErrorCode)
          return (ErrorCode);

     Opened = 1;                /* Set opened flag.                          */

     return (0);

     } /* end _PrtOpen */
```

```
USHORT _StrLen
     (char far *String)
/*
     This private function returns the length of the NULL-terminated string
     'String'.
*/
     {
     USHORT Count = 0;

     while (*String++)
          ++Count;

     return (Count);

     } /* end _StrLen */
```

Figure 9-3

```
;    Figure 9-3
;
;    This file defines the assembly language functions for the real-mode
;    version of the module of printer functions.  The C functions are defined
;    in Figure 9-2.  The real-mode library can be prepared using the MAKE file
;    of Figure 9-4.
;
.MODEL LARGE
.CODE

EXTRN DOSSLEEP:FAR              ;External declaration for DosSleep API function.
```

```
Frame     equ  [bp]           ;Equate for accessing parameters.

PUBLIC PRTREADY
;
;    Prototype:
;         unsigned pascal far PrtReady
;              (unsigned char far *PtrFlagReady);
;
PRTREADY PROC FAR

prFrame        struc          ;Template for accessing the parameters.
prBasePtr      dw   ?
prRetAd        dd   ?
prFlagAddr     dd   ?         ;Parameter 'PtrFlagReady'.
prFrame        ends

     push bp                  ;Save base pointer.
     mov  bp, sp              ;Set base pointer to access stack.

     mov ah, 2                ;Invoke BIOS printer services function 2: read
     mov dx, 0                ;status of printer port.
     int 17h

     and ah, 10h              ;Results are returned in AH;  mask all bits of
                              ;status except 'selected' bit.

     les bx, Frame.prFlagAddr ;Assign results to the memory location whose
     mov byte ptr es:[bx], ah ;address was passed in the parameter
                              ;'PtrFlagReady'.


     xor ax, ax               ;Set return value in AX to 0.
```

```
        pop bp                  ;Restore base pointer.
        ret 4                   ;FAR return / remove parameter from stack.

PRTREADY ENDP

PUBLIC PRTINIT
;
;       Prototype:
;           unsigned pascal far PrtInit
;                (void)
;
PRTINIT PROC FAR

        mov ah, 1               ;Invoke BIOS printer services function 1:
        mov dx, 0               ;initialize printer port.
        int 17h

                                ;Generate a 1.5 second pause:
        xor ax, ax              ;Push a value of 1500L on the stack.
        push ax
        mov ax, 1500
        push ax
        call DOSSLEEP           ;Call API function DosSleep.

        xor ax, ax              ;Set return value in AX to 0.
        ret                     ;FAR return instruction.

PRTINIT ENDP

END
```

The definitions of the printer functions found in these two figures illustrate the following general guidelines for developing real-mode versions of your dynamic-link functions:

1. Use the OS/2 family API functions.
2. Observe the real-mode restrictions on family API functions.
3. Do not use the C runtime library.
4. Write your code specifically for real mode.
5. Compensate for differences between real and protected modes.

These guidelines are now discussed individually.

Use the OS/2 Family API Functions

The source file of Figure 9-2 defines the following public functions:

PrtPutC
PrtPutS
PrtPosition
PrtNewPage

Note that these functions are defined in the same manner as in the C source file of Figure 4-2. You might think that when creating real-mode versions of dynamic-link functions, you would have to eliminate all calls to OS/2 API services. Fortunately, however, this is not necessary since your function code can take advantage of the real-mode versions of the API functions (contained in API.LIB) in the same manner as an application program that is converted to the dual-mode format. Of course, you must restrict your API calls to the family subset.

Observe Real-Mode Restrictions on Family API Functions

Two of the printer functions in the dynamic-link version of the printer module of Figure 4-2—PrtReady and PrtInit—call the OS/2 function DosDevIOCtl to send control information to the printer. Although DosDevIOCtl is a member of the family API, in real mode it does not support the printer control functions (that is, the functions in category 5).

Accordingly, the real-mode versions of these two functions had to be rewritten to eliminate the use of DosDevIOCtl.

The new versions of these functions obtain the required printer services by directly invoking the BIOS services provided in real mode through interrupt 17h.

Do Not Use the C Runtime Library

Normally, a real-mode C program could invoke interrupt 17h through a C library function such as **int86**. However, attempting to use the C runtime library causes problems when processing your real-mode library with the BIND utility.

You could include a real-mode version of the C library (such as SLIBCER.LIB) among the libraries processed by BIND (passing multiple library files to BIND is explained in the next section). However, calling a C library function creates an external reference to the function main, which is normally not defined within your real-mode library or within any of the other real-mode libraries processed by BIND. Also, by binding in C library code you would be attempting to execute real-mode versions of the C library functions without running the real-mode C startup code.

To avoid these problems, the example real-mode library eliminates the need for C runtime functions by providing low-level assembly language routines.

Write Your Code Specifically for Real Mode

As you saw when developing a dual-mode program, you must make sure that your code can run in either real or protected mode. Consequently, you must not include instructions that are prohibited in protected mode, such as software interrupt instructions and accesses to absolute memory locations outside of the segments allocated to the process unless you test the machine mode and issue the instructions only if the machine is in real mode.

In contrast, when developing real-mode versions of dynamic-link functions, your code need run only in real mode. Consequently, the assembler functions in Figure 9-3 freely invoke BIOS services through software interrupt instructions.

Differences between Real and Protected Modes

Finally, there may be subtle differences in the behavior of the protected-mode API functions and the equivalent real-mode functions. For example, when attempting to write to a busy printer using the protected-mode version of DosWrite, the system waits for a timeout period before generating a critical-error message. However, when using the real-mode version of this function, the system immediately activates a critical error.

Consequently, the real-mode version of the PrtInit function in Figure 9-3 inserts a 1.5 second delay. This delay is required since the printer is temporarily busy during a reset operation; if the calling program attempted to write to the printer immediately after calling PrtInit, the system would generate a critical-error message. Inclusion of the delay allows the reset operation to complete before PrtInit returns control.

In general, you should thoroughly test your functions under both protected and real modes to detect any differences in behavior between these two modes.

Using Real-Mode Versions

Figure 9-4 provides a MAKE file that shows how to use the real-mode versions of a module of dynamic-link functions to generate a dual-mode application program. This MAKE file creates a dual-mode executable file from the program given in Figure 4-7 (remember that when this program was prepared in Chapter 4, the resulting file could be run only in protected mode).

- The MAKE file generates the following primary files:
- The dynamic-link library containing the printer functions (FIG4_2.DLL; from the module source code of Figure 4-2).
- The associated import library (FIG4_2.LIB).
- The library file containing real-mode versions of the dynamic-link printer functions (FIG9_2.LIB; from the source code of Figure 9-2).
- The dual-mode executable program (FIG4_7.EXE; from the C program of Figure 4-7).

Figure 9-4

```
#    Figure 9-4
#    A MAKE script for preparing the following files:
#
#    1.  The dynamic-link library file FIG4_2.DLL
#    2.  The real-mode library FIG9_2.LIB
#    3.  The dual-mode application program FIG4_7.EXE.

# Produce the dynamic-link library:
FIG4_2.OBJ : FIG4_2.C FIG4_3.H
     cl /c /W2 /ASw /G2s /Zp FIG4_2.C

FIG4_2.DLL : FIG4_2.OBJ FIG4_4.DEF
     link /NOI /NOD FIG4_2.OBJ, FIG4_2.DLL, NUL, OS2.LIB, FIG4_4.DEF

# Produce the import library:
FIG4_2.LIB : FIG4_4.DEF
     implib FIG4_2.LIB FIG4_4.DEF

# Produce the real-mode library file:
FIG9_2.OBJ : FIG9_2.C FIG4_3.H
     cl /c /W2 /Gs /Zp FIG9_2.C


FIG9_3.OBJ : FIG9_3.ASM
     masm /MX FIG9_3.ASM;


FIG9_2.LIB : FIG9_2.OBJ FIG9_3.OBJ
     lib FIG9_2.LIB -+FIG9_2.OBJ -+FIG9_3.OBJ;
```

```
# Produce the dual-mode client application:

FIG4_7.OBJ : FIG4_7.C FIG4_3.H
     cl /c /W2 /Zp FIG4_7.C

FIG4_7.EXE : FIG4_7.OBJ FIG9_2.LIB
     link /NOI /NOD FIG4_7.OBJ,, NUL, FIG4_2.LIB SLIBCE.LIB OS2.LIB;
     bind FIG4_7.EXE \PMSDK\LIB\OS2.LIB FIG4_2.LIB \PMSDK\LIB\API.LIB \
          FIG9_2.LIB
```

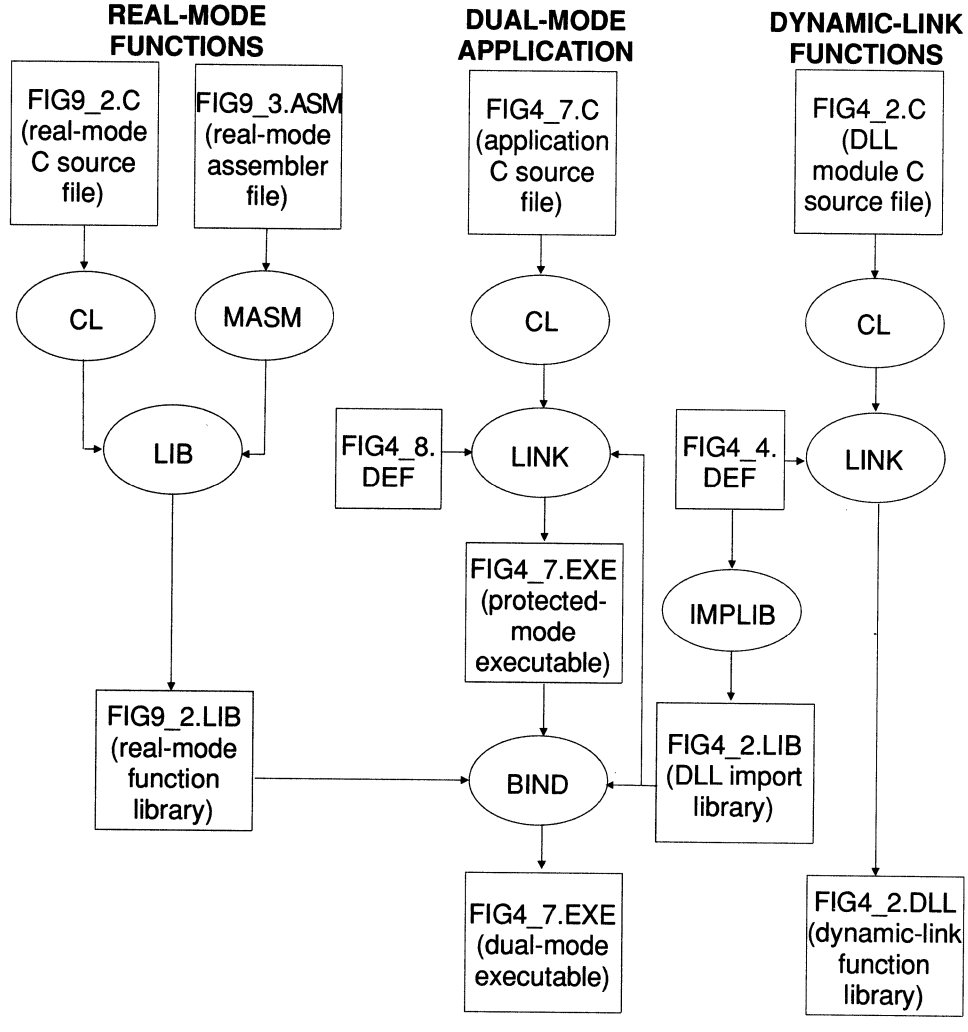
The relationships among the files processed by this MAKE script are illustrated in Figure 9-5.

Note that when compiling the real-mode functions defined in Figure 9-2, the MAKE file does not include the /ASw flag (explained in Chapter 1). This flag is not needed since during the execution of the real-mode functions—unlike their dynamic-link counterparts—the SS and DS registers contain the same segment value.

Note also that the **/G2** flag which allows the compiler to use instructions specific to the 80286 and later-model processors is not included when compiling the real-mode functions (FIG9_2.C) or the dual-mode program (FIG4_7.C). This flag is eliminated since a dual-mode program can be run under MS-DOS on a machine with an 8086 or 8088 processor.

Note finally that when invoking BIND, the MAKE file includes both API.LIB (to supply real-mode versions of the OS/2 API functions) and FIG4_2.LIB (to supply real-mode versions of the dynamic-link printer functions). Also, it specifies the import library FIG4_2.LIB, originally used to link the application program; the import library is required when an application program identifies the dynamic-link functions by their ordinal values. Remember that the EXPORT statement in the module definition file of Figure 4-4 specified the ordinal value of each dynamic-link function.

Figure 9-5: Operations performed by the MAKE script of Figure 9-4.



After you have entered the required source files and have processed them with the MAKE script of Figure 9-4, you should be able to run the resulting program (FIG4_7.EXE) within any of the following four environments:

- From the OS/2 command line of a protected-mode screen group.
- Within a window of the Presentation Manager.
- From the command line of the OS/2 DOS compatibility environment.
- Under MS-DOS.

CHAPTER 10

ASSEMBLY LANGUAGE DLLS

The OS/2 application program interface is strongly oriented toward high-level languages. As you have seen, the dynamic linking mechanism allows programs in high-level languages to call API functions using the standard function calling protocol. Also, high-level compilers such as Microsoft C automatically generate code that conforms to most of the requirements of the protected mode (such as storing code and data in separate segments and not writing to code segments).

In contrast, calling API functions from assembly language is somewhat cumbersome. Also, the assembly language programmer must take care to write code compatible with a protected-mode operating system; many of the low-level tactics typically performed in assembly language are prohibited in such a system.

Assembly language, however, is important for writing several types of special-purpose routines under OS/2. Among these routines are the following:

- **Initialization routines**. As you saw in Chapter 6, you must write an assembly language module to define the initialization entry point.

- **Real-mode versions of dynamic-link functions.** As described in Chapter 9, you may need to write an assembly language routine to replace an OS/2 service that is not supported in real mode.
- **I/O privileged routines.** You will see in this chapter that assembly language is required to develop privileged routines that access I/O ports or control interrupts.

The first section of this chapter explains the basic techniques for writing a dynamic-link module in assembly language; as an example, it presents an assembly language version of the dynamic-link function PrtPutC (which is defined in C in Figure 4-2). The second section explains an important use for assembly language: writing dynamic-link functions that execute with I/O privilege.

This chapter is not a primer on assembly language programming; rather, it focuses on the special techniques required when using assembly language to develop dynamic link libraries. For basic information on programming in assembly language, see the documentation provided with the Macro Assembler (now much improved), or the introductory book by Lafore cited in the Bibliography. See also the Microsoft *Mixed Language Guide* (included with the high-level language compilers) for more information on interfacing assembly language routines and C programs.

General Guidelines for Assembly Language

Figure 10-1 provides an example of a dynamic link module written in assembly language. This module defines a single dynamic-link function: PrtPutC, which is an assembly language version of the PrtPutC function defined in the C file of Figure 4-2. Figure 10-2 lists the module definition file, and Figure 10-3 contains the C code for an example program that calls PrtPutC. Figure 10-4 provides a MAKE script for preparing both the dynamic-link library and the example application. Each of these files is now discussed individually.

Figure 10-1

```
;     This file defines an assembly language version of the function 'PrtPutC'
;     (the C version of this function is contained in Figure 4-2). You can use
;     the MAKE script given in Figure 10-4 to prepare the dynamic-link library
;     file, as well as the example client application of Figure 10-3.
;

.286                            ;Allow 80286 instructions.

.MODEL LARGE                    ;Large memory model.

.DATA                           ;Begin automatic data segment.

Handle     dw   0               ;File handle for printer received from 'DosOpen'.
Action     dw   0               ;Receives action taken by 'DosOpen'.
DevName    db   'LPT1',0        ;Device name for printer.
ByWritten  dw   0               ;Receives bytes written by 'DosWrite'.

.CODE                           ;Begin code segment.

                                ;External declarations for OS/2 API functions:
EXTRN DOSOPEN:FAR               ;'DosOpen'.
EXTRN DOSWRITE:FAR              ;'DosWrite'.

InitRout PROC FAR
                                        ;Push parameters for call to 'DosOpen':
     push ds                            ;Address of device name string.
     push offset DGROUP:DevName

     push ds                            ;Address to receive device handle.
     push offset DGROUP:Handle

     push ds                            ;Address to receive code for action taken.
     push offset DGROUP:Action
     push 0                             ;Initial allocation size (long value):  n/a.
```

```
        push 0
        push 0                          ;File attribute:  n/a.

        push 1                          ;Open flag:  open file if it exists.

        push 0041h                      ;Open mode:  share/write-only.

        push 0                          ;Reserved double word value:  must be 0.
        push 0

        call DosOpen                    ;Open the printer device.

        or   ax, ax                     ;Test error status in AX.
        jz   iOK                        ;0 indicates 'DosOpen' was successful.

        xor  ax, ax                     ;'DosOpen' failed, therefore return 0, which
        ret                             ;indicates that the initialization routine
                                        ;failed.
iOK:
        mov ax, 1                       ;'DosOpen' was successful, therefore return
        ret                             ;1, indicating that the initialization
                                        ;routine was successful.

InitRout ENDP
PUBLIC PRTPUTC                  ;Define the dynamic-link function 'PrtPutC'.
;
;    Prototype:
;         unsigned pascal far _loadds PrtPutC
;              (int Ch);
;
PrtPutC PROC FAR
pcFrame         struc           ;Template for accessing the parameter.
pcBasePtr       dw   ?
pcRetAd         dd   ?
```

```
pcCh            dw   ?          ;Parameter 'Ch'.
pcFrame         ends

Frame equ [bp]                  ;Equate for accessing parameters.

     push bp                    ;Save base pointer.
     mov  bp, sp                ;Set base pointer to access stack.
     push ds                    ;Save DS register.

     mov  ax, DGROUP            ;Assign to DS register the segment selector of
     mov  ds, ax                ;the automatic data segment.

                                ;Push parameters for call to 'DosWrite':
     push Handle                ;Printer device handle.

     push ss                    ;FAR address of 'Ch', which is on the stack.
     lea  ax, Frame.pcCh
     push ax

     push 1                     ;Length of the buffer ('Ch').

     push ds                    ;Address to receive number of bytes written.
     push offset DGROUP:ByWritten

     call DosWrite              ;Send the character to the printer.

     pop  ds                    ;Restore DS.
     pop  bp                    ;Restore base pointer.
     ret  2                     ;FAR return instruction;  value returned in AX is
                                ;the error code supplied by 'DosWrite';  also
                                ;remove parameter from stack.
PrtPutC ENDP

END InitRout                    ;Terminate source file and define the
                                ;initialization entry point.
```

Figure 10-2

```
;               Figure 10-2
;
;               A module definition file for linking the dynamic-link
;               library defined in Figure 10-1.
;
LIBRARY FIG10_1 INITINSTANCE

PROTMODE

DATA MULTIPLE

EXPORTS
                PRTPUTC@1
```

Figure 10-3

```
/*
     Figure 10-3

     This example client program calls the dynamic-link function 'PrtPutC',
     defined in the assembly language file of Figure 10-1.  You can use the
     MAKE file of Figure 10-4 to prepare this program, as well as the
     dynamic-link library.
*/

#include <STDIO.H>
#include <CONIO.H>
```

```
#include <PROCESS.H>
unsigned pascal far _loadds PrtPutC      /* Declare dynamic-link function.     */
     (int Ch);

void main (void)
     {
     int Ch;

     printf ("Type characters to print ... Press Esc to quit.\n");
     while ((Ch = getche ()) != '\x1b')
          if (PrtPutC (Ch))
               {
               fprintf (stderr, "\nError writing to printer.");
               exit (1);
               }

     } /* end main */
```

Figure 10-4

```
#     Figure 10-4
#     A MAKE file for preparing the dynamic-link module of Figure 10-1, and the
#     client program of Figure 10-3.
#

# Prepare the dynamic-link library and import library:
FIG10_1.OBJ : FIG10_1.ASM
     masm /MX FIG10_1.ASM;

FIG10_1.DLL : FIG10_1.OBJ FIG10_2.DEF
     link /NOI FIG10_1.OBJ, FIG10_1.DLL, NUL.MAP, OS2.LIB, FIG10_2.DEF
```

```
FIG10_1.LIB : FIG10_2.DEF
     implib FIG10_1.LIB FIG10_2.DEF

# Prepare the client program:
FIG10_3.OBJ : FIG10_3.C
     cl /c /W2 /G2 /Zp FIG10_3.C

FIG10_3.EXE : FIG10_3.OBJ FIG10_1.LIB
     link /NOI /NOD FIG10_3.OBJ,, NUL.MAP, FIG10_1.LIB SLIBCE.LIB OS2.LIB;
```

The Assembly Language Source Code

The assembly language file of Figure 10-1 begins with the directive **.286**. This directive allows you to use instructions unique to the 80286 processor. You can freely issue these instructions since OS/2 requires an 80286 or later-model processor. As an example, the 80286 processor allows you to push immediate values on the stack, that is, you can specify constant numeric values with the PUSH instruction; under the 8088 processor, you must specify either a register or a memory variable. As you will see, calling API functions from assembly language requires you to push many values on the stack. The file next contains the directive:

```
.MODEL LARGE
```

This is one of several simplified segment directives offered by Microsoft MASM beginning with version 5.0 (see the assembler documentation for complete information on the use of the new simplified segment directives). This directive instructs the assembler to use—by default—segment names and other segment attributes that match those employed in a large memory model Microsoft C (or other high-level language) program. It replaces the ASSUME and GROUP statements required with former versions of the assembler.

Since the example dynamic-link module contains only a single code segment and a single data segment, the large memory model is not required. In general, however, it is the most flexible memory model. Also, under the large model, the compiler automatically generates the required FAR return instructions. Even if you forget to specify the FAR attribute when defining a procedure; remember that a dynamic-link function must terminate with a FAR return instruction.

The assembly language file uses two additional simplified segment directives: .DATA and .CODE (note that if you use these directives, the .MODEL directive is required, and must be placed *before* the occurrence of either .DATA or .CODE). These two directives replace the SEGMENT and ENDS statements required under former versions of the assembler.

The .DATA directive defines the beginning of the data segment. Specifically, it causes the assembler to create a logical segment named _DATA, which is normally used in Microsoft high-level languages for storing initialized data. This logical segment is placed within the automatic data segment described in Chapter 5. The assembly language module defines several variables within this segment, which are used to exchange information with OS/2 API functions. For example, the segment contains the variable *Handle*, which stores the device handle to the printer. This handle is obtained by the initialization routine and is then used to write to the printer each time PrtPutC is called.

The .CODE directive defines the beginning of the module code segment. This directive automatically closes the segment opened with the .DATA directive (no ENDS statement is required). The code segment contains the two procedures defined within the module: InitRout and PrtPutC.

InitRout is the dynamic-link library initialization routine (see Chapter 6 for a general description of initialization routines). InitRout is specified as the module entry point by the END statement at the conclusion of the file.

InitRout calls the OS/2 API function DosOpen to open the printer device and to obtain a valid device handle. A device (or file) handle is valid only within the current process. Therefore, the initialization procedure must be made an **instance routine**—it must be called when first referenced by each new client process—and the variable used to store the handle (*Handle*) must be contained in an **instance data segment**. As you will

see, both of these instance attributes are set by the module definition file of Figure 10-2.

Using an initialization routine to obtain a printer handle obviates the need to test whether the printer has been opened each time PrtPutC is called. Such a test is necessary in the C version of PrtInit in Figure 4-2, since the module defined by this file does not have an initialization routine.

The call to DosOpen in the initialization routine illustrates the following four guidelines for calling an OS/2 API function from an assembly language procedure (compare this function call with the call to DosOpen in the C source file of Figure 4-2).

1. You must declare the API service as an external function that is invoked through a far call, as in this statement from Figure 10-1:

   ```
   EXTRN DOSOPEN : FAR
   ```

   Note that the name of the API function must be given in uppercase letters; the reason for this requirement is explained later in the chapter (in the section The MAKE File).

2. Immediately before calling the function, you must push each of the parameters on the stack. Since the API functions use the Pascal calling conventions, the parameters must be pushed on the stack in the same order that they are listed in the function prototype. For example, the following code from Figure 10-1 pushes the parameters and calls DosOpen:

```
push ds                          ;Address of device name string.
push offset DGROUP:DevName

push ds                          ;Address to receive device handle.
push offset DGROUP:Handle

push ds                          ;Address to receive code for action taken.
push offset DGROUP:Action

push 0                           ;Initial allocation size: n/a.
push 0
```

```
push 0                          ;File attribute: n/a.

push 1                          ;Open flag: open file if it exists.

push 0041h                      ;Open mode: share/write-only.

push 0                          ;Reserved double word value: must be 0.

push 0

call DosOpen                    ;Open the printer device.
```

When an initialization routine is invoked, register DS already contains the selector of the automatic data segment (provided that the module has defined such a segment). Thus, DS can be used to push the selector portion of the far addresses of variables contained within this segment (when passing a far address, you must first push the segment value and then the offset). The assembler allows you to push immediate numeric values on the stack because of the .286 directive given at the beginning of the file.

3. You do not have to restore the stack pointer after the API function returns. Since these functions use the Pascal calling protocol, the functions themselves restore the stack (using the RET *n* form of the return instruction, explained later in the chapter).

4. The API functions return the error status in register AX. As you have seen, if an API function is successful, it returns zero, and if it fails, it returns a nonzero error code. Accordingly, if AX contains a nonzero value after calling DosOpen, the initialization routine returns *zero* in register AX, which notifies the system that the initialization routine has failed, thus, the meaning of a zero return value from the initialization routine is the opposite of the meaning of a zero return value from an API function. As mentioned in Chapter 6, when the system receives a zero return value from the initialization routine, it summarily aborts the loading procedure.

Microsoft provides a set of OS/2 header files for code written in assembly language. These files have the .INC extension and can be incorporated in the source file through the INCLUDE assembler directive. Among the items contained in these files is a collection of macros that can simplify calling API functions; when using these macros, the syntax for calling API functions resembles calls made from high-level languages. In order to illustrate basic programming techniques, the example assembly language source files given in this book do not use these macros. See the technical documentation and the INCLUDE files for information on using these macros in your OS/2 assembly language code.

PrtPutC is the only public function defined in the assembly language file of Figure 10-1. This function simply calls the API function DosWrite to print the character that is passed as a parameter. This function call is similar to the call to DosOpen. Note that PrtPutC—unlike its C counterpart defined in Figure 4-2—does not need to test for a valid printer handle (and open the printer if necessary), since the printer is always opened by the initialization routine. The initialization routine, of course, always receives control before PrtPutC can be called.

The definition of PrtPutC in Figure 10-1 illustrates the following six important techniques for writing dynamic-link functions in assembly language:

1. **Declaring the function as public**. To allow the function to be exported (that is, made available to client programs), the function name is declared as a public symbol as follows:

   ```
   PUBLIC PRTPUTC
   ```

   When specified in a PUBLIC statement, the function name must be written in all uppercase letters; the reason will be explained later in the chapter (in the section The MAKE File).

2. **Saving registers**. PrtPutC saves the required register values. An assembly language routine called by a Microsoft C function must preserve the values of the following registers: SI, DI, SP, BP, DS, CS, and SS. Of these registers, PrtPutC, modifies only BP, DS, and SP. Therefore, it pushes BP and DS on the stack at the beginning of the function and pops them back off the stack immediately before the function returns. The value of the stack pointer is automatically preserved

since there are an equal number of PUSH and POP instructions during the execution of the function.

3. **Accessing the stack**. To access its parameter on the stack, PrtPutC conforms to the standard practice of assigning the stack pointer value (SP) to the base pointer register (BP) immediately after saving the original value of BP on the stack. To facilitate addressing this parameter, it also defines a structure (**pcFrame**) with an element corresponding to each item stored in the stack frame. This structure merely serves as a template for accessing the stack; it does not reserve memory. PrtPutC also defines the symbol Frame (equal to [bp]). As a result, this function can access its parameter using the straightforward structure notation—**Frame.phCh**. This technique for accessing the stack eliminates the need to calculate the position of each parameter on the stack; it is especially valuable for functions that have several parameters of various data types. If the function has several parameters, these parameters should be listed in the structure in the same order that they appear in the prototype for a normal C function, and they should be listed in the reverse order that they appear in the prototype for a pascal function.

4. **Loading DS**. When a dynamic-link function receives control, the DS register typically contains the selector of the client's automatic data segment. Therefore, to allow the function to access variables within its own data segment, the DS register must be assigned the selector of the module's own data segment. This value is represented by the symbol DGROUP. When using the simplified segment directives .MODEL and .DATA, the assembler automatically assigns the data segment to the DGROUP segment group; the symbol DGROUP stands for the selector value of the physical segment containing the logical segment or segments belonging to this group. A C function loads the DS register in this same manner when you define the function using the _loadds keyword, or when you compile the program under the /Au flag.

5. **Restoring the stack**. The function should return with the RET *n* form of the return instruction, where *n* is the number of bytes that should be removed from the stack upon function return. This form of the command is necessary for any dynamic-link function that is defined as pascal. In contrast, when a non-pascal function is called, the calling program is responsible for restoring the stack.

6. **Returning a value**. An assembly language procedure returns an integer value to a C function by assigning it to register AX. PrtPutC, however, does not explicitly assign a value to AX; rather, when PrtPutC returns, AX contains the error code value that was assigned to this register by DosWrite. Accordingly, PrtPutC (in the same manner as the C version of this function) returns the error status supplied by DosWrite.

The Module Definition File

Figure 10-2 lists the module definition file used to link the example dynamic-link library. This file specifies the INITINSTANCE option under the LIBRARY statement, which defines the module entry point (INITROUT) as an instance initialization routine, that is activated for each new client. The definition file also includes the DATA MULTIPLE statement to make the data segment an instance segment. Finally, it exports the dynamic- link function PrtPutC using the usual EXPORTS statement.

The Client Program

Figure 10-3 lists a simple client program for demonstrating the use of the dynamic-link function PrtPutC. This program acts as a typewriter; it calls PrtPutC to print each character entered from the keyboard, until the user presses Esc.

The MAKE File

Figure 10-4 provides a MAKE file for preparing both the dynamic-link library defined in Figure 10-1 and the client program of Figure 10-3. In addition to the usual set of flags, which have already been explained, the MAKE file specifies the /MX flag in the assembler command line.

The /MX option causes the assembler to preserve the case in all names specified in EXTERN and PUBLIC statements. These two statements list the names that are referenced in other modules (for example, other assembly languages or C source files). The EXTRN statement declares names that are defined in other modules but are used in the current module (such as DosOpen). The PUBLIC statement declares names that

are defined in the current module but are used in other modules, such as PrtPutC.

By default, the assembler converts the names you list in these statements to all uppercase letters. If, however, you specify the /MX option, the case is preserved when the symbols are written to the object file. Since the linker is instructed to distinguish upper and lowercase letters (through the /NOI flag), the names you specify in PUBLIC and EXTRN statements must match exactly—including the case of all letters—the corresponding names contained in all other object modules. Consequently, the names of all pascal functions (such as DosOpen and PrtPutC) must be given in uppercase letters, while the names of normal C variables and functions must be written with the exact combination of uppercase and lowercase letters used in the C source file, with the addition of a leading underscore character (as seen in Chapter 6).

Finally the LINK command line in this MAKE file—unlike those seen previously—does not specify the /NOD option, since the assembler does not normally insert the names of default search libraries into the object files it generates.

I/O Privileged Dynamic-Link Functions

Figure 10-5 lists an assembly language source file, which defines a dynamic-link function that executes with I/O privilege. Figure 10-6 contains the module definition file, and Figure 10-7 gives an example client program. You can prepare both the dynamic-link library and the client program using the MAKE file of Figure 10-8.

Figure 10-5 defines the dynamic-link function **HercCard**, which reports whether a Hercules graphics card is installed in the machine. This function has the following prototype:

```
unsigned pascal far HercCard
     (int far *PPresent)
```

HercCard assigns the variable pointed to by the parameter **PPresent** a nonzero value if a Hercules card is present, and zero if the card is not present.

Figure 10-5

```
;     Figure 10-5
;
;     This file defines the dynamic-link function 'HercCard', which reports
;     whether a Hercules graphics card is installed in the machine. This
;     function runs as an I/O privileged routine. The dynamic-link library can
;     be prepared using the MAKE file of Figure 10-8.

.286                              ;Allow 80286 instructions.

.MODEL LARGE                      ;Large memory model.

.CODE                             ;Begin code segment.

EXTRN DOSPORTACCESS:FAR           ;OS/2 'DosPortAccess' function.

PUBLIC HERCCARD                   ;Define the dynamic-link function 'HercCard'.
;
;     Prototype:
;          unsigned pascal far HercCard
;               (int far *PPresent);
;
HercCard PROC FAR

hcFrame         struc             ;Template for accessing the parameters.
hcBasePtr       dw    ?
hcRetAd         dd    ?
hcPPresent      dd    ?           ;Parameter 'PPresent'.
hcFrame         ends

Frame equ [bp]                    ;Equate for accessing parameter.

     push bp                      ;Save base pointer.
```

```
    mov  bp, sp              ;Set base pointer to access stack.

     ;--- Obtain port access. ------------------------------------------

                              ;Push parameters for call to 'DosPortAccess'.
     push 0                   ;Reserved value:  must be 0.

     push 0000h               ;Desired function:  grant access to port.

     push 03bah               ;First port in range:  0x03ba.

     push 03bah               ;Last port in range:   0x03ba.

     call DosPortAccess       ;Request port access.

     xor  ax, ax              ;Test for error.
     jz   ok1
     jmp  exit                ;Error occurred;  return error code in AX.
ok1:                          ;No error.

     ;--- Test for presence of Hercules Graphics Card ----------------------
                              ;This routine supplied courtesy of
                              ;Hercules Computer Technology.
     mov  dx, 03bah           ;Display status port.
     xor  bl, bl              ;Clear counter.
     in   al, dx              ;Read port.
     and  al, 80h             ;Mask off all bits except 7.
     mov  ah, al              ;Save bit 7 in AH.
     mov  cx, 8000h           ;Set loop counter.
a01:
     in   al, dx              ;Read port again.
     and  al, 80h             ;Mask out bit 7.
```

```
        cmp   al, ah                ;Test if bit has changed.
        je    a02                   ;Bit not yet changed.
        inc   bl                    ;Bit changed, increment counter.
        cmp   bl, 10                ;Want to see it change 10 times.
        jb    a02                   ;Need to see more changes.
        mov   ax, 1                 ;Yes, it is a HGC.
        jmp   a03                   ;Go to end.
a02:
        loop  a01                   ;Continue testing for changes.
        xor   ax, ax                ;Hercules card is not present.
a03:
                                    ;Write result in AX to memory location
                                    ;pointed to by the parameter 'Present'.
        les   bx, Frame.hcPPresent
        mov   es:[bx], ax

        ;--- Release port access ------------------------------------------

                                    ;Push parameters for call to 'DosPortAccess'.
        push 0                      ;Reserved value: must be 0.

        push 0001h                  ;Desired function:  release access to port.

        push 03bah                  ;First port in range:  0x03ba.
        push 03bah                  ;Last port in range:   0x03ba.

        call DosPortAccess          ;Release port access.

        xor   ax, ax                ;Test for error.
        jz    ok2
        jmp   exit                  ;Error occurred;  return error code in AX.
ok2:                                ;No error.
```

```
     ;--- Set return value and return to caller ----------------------------

     xor  ax, ax              ;Set error code returned in AX to 0 to indicate
                              ;that function was successful.
exit:
     pop  bp                  ;Restore base pointer.
     ret  4                   ;FAR return / remove parameters from stack.

HercCard ENDP

END
```

Figure 10-6

```
;              Figure 10-6
;
;              A module definition file for creating the dynamic-link
               library defined in Figure 10-5.
;
LIBRARY FIG10_5

PROTMODE

CODE IOPL

EXPORTS
               HERCCARD       @1      2
```

Figure 10-7

```
/*
     Figure 10-7

     This example client program calls the dynamic-link function 'HercCard' to
     determine whether a Hercules Graphics Card is installed in the machine.
     'HercCard' is defined as an I/O privileged routine in the assembly
     language source file of Figure 10-5.  You can use the MAKE script of
     Figure 10-8 to prepare both the dynamic-link library and the program
     defined in the present file.
*/

#include <STDIO.H>
#include <PROCESS.H>

unsigned pascal far HercCard            /* Declare dynamic-link function.    */
     (int far *PPresent);

void main (void)
     {
     int Present;

     if (HercCard (&Present))
          {
          fprintf (stderr, "Error calling the 'HercCard' function.\n");
          exit (1);
          }

     printf ("A Hercules Graphics Card is %spresent.\n", Present ? "" : "not");

     } /* end main */
```

Figure 10-8

```
#     Figure 10-8
#     A MAKE file for preparing the dynamic-link module of Figure 10-4, and
#     the client program of Figure 10-7.
#

# Prepare the dynamic-link library and import library:
FIG10_5.OBJ : FIG10_5.ASM
     masm /MX FIG10_5.ASM;

FIG10_5.DLL : FIG10_5.OBJ FIG10_6.DEF
     link /NOI FIG10_5.OBJ, FIG10_5.DLL, NUL.MAP, OS2.LIB, FIG10_6.DEF

FIG10_5.LIB : FIG10_6.DEF
     implib FIG10_5.LIB FIG10_6.DEF

# Prepare the client program:
FIG10_7.OBJ : FIG10_7.C
     cl /c /W2 /G2 /Zp FIG10_7.C

FIG10_7.EXE : FIG10_7.OBJ FIG10_5.LIB
     link /NOI /NOD FIG10_7.OBJ,, NUL.MAP, FIG10_5.LIB SLIBCE.LIB OS2.LIB;
```

Following the conventions of the OS/2 API, the function directly returns the error status to the calling program. If the function is successful, it returns zero; if it fails, it returns the nonzero error code returned by the API function that caused the failure.

This function might be part of a dynamic-link library of video routines. It determines the presence of a Hercules card by testing whether bit 7 of

the **display mode status port** (I/O port number 0x03ba) changes at least 10 times during the execution of a loop of 8000h repetitions. It is a unique characteristic of the Hercules Graphics Card that this bit goes low with each vertical retrace. The routine is based upon an algorithm supplied by Hercules Computer Technology. The details of the routine are documented in the listing; the important feature is that the function must read an I/O port. To be able to access a port, a routine must conform to the following two general requirements:

1. The routine must be written in assembly language, since the protected-mode version of the C runtime library does not provide functions for reading or writing to ports. The C functions **inp**, **inpw**, **outp**, and **outpw** are provided only in the real-mode version of the library).

2. The routine must execute with I/O privilege, which permits it to perform direct access to I/O ports.

A normal OS/2 application program is not allowed to execute the (OS/2 version 1.1) machine instructions listed here in Table 10-1:

Table 10-1: Machine Instructions

INSTRUCTION	PURPOSE
IN	Read a byte or word from an input port.
INS	Read a string from an input port.
OUT	Write a byte or word to an output port.
OUTS	Write a string to an output port.
CLI	Disable hardware interrupts.
STI	Enable hardware interrupts.

Attempting to execute any of these instructions from a normal program or dynamic-link library will cause a protection fault. However, a routine that executes with I/O privilege—one contained in an I/O privileged code

segment is permitted to use any of these instructions. However, that having I/O privilege does not allow a function to use all machine instructions provided by the processor. There is a set of restricted instructions that can be issued only by the operating system kernel, and is off-limits to both normal routines as well as I/O privileged code. Both normal and I/O privileged routines are also prohibited from servicing hardware interrupts and from executing software interrupt instructions.)

The example listings given in this section illustrate five basic requirements for creating a function that executes with I/O privilege. Although the example I/O privileged routine contained in a dynamic-link library, you can also follow these guidelines to grant I/O privilege to a function within an application program. The example module illustrates these requirements as follows:

1. The module definition file (Figure 10-6) contains the statement:

```
CODE IOPL
```

which causes the linker to mark the dynamic-link module code segment (which contains the function HercCard) so that it will be executed with I/O privilege. Therefore, that I/O privilege is a property of a given code segment. Any function contained in a code segment that is designated as I/O privileged will execute with a privilege level that allows it to use the restricted machine instructions listed above.

Also the CODE statement applies globally to all code segments in a particular dynamic-link library or program. If your dynamic-link library or program defines functions located in more than one code segment, you can assign I/O privilege to a specific segment (or segments) using the SEGMENTS command. For example, if a dynamic-link library contains two code segments—named MOD1_TEXT and MOD2_TEXT—you can grant I/O privilege only to MOD2_TEXT through the following statement:

```
SEGMENTS MOD2_TEXT IOPL
```

Note finally that a dynamic-link initialization routine (discussed in Chapter 6) must not run with I/O privilege. If your dynamic-link module defines an initialization routine, you can use the SEGMENTS statement to grant I/O privilege to a code segment other than the one containing the

initialization code. In this case, do not use the CODE statement, which would automatically grant I/O privilege to all code segments.

2. The dynamic-link function HercCard is declared as a pascal function in the calling program of Figure 10-7, and the function definition in Figure 10-5 conforms to the Pascal calling convention. It returns with the RET 4 machine instruction to restore the stack). Two specific features of the Pascal calling convention are required to be able to call an I/O privileged routine from non-privileged code. First, the routine must be passed a fixed number of parameters; second, the function itself must restore the stack. Both these features are required because a function in an I/O privileged code segment is called from a non-privileged segment through a **call gate**—as explained later in this section.

3. HercCard is defined as a far function. A function contained in an I/O privileged segment that is called from a function in a non-privileged segment must be defined as far. A far call is required because it is called from a separate segment, and because the call passes through a call gate. A function contained in an I/O privileged segment need be declared as pascal far only if it is called from a non-privileged code segment. A local function, which is called only from other functions within the same privileged code segment, can use standard C calling conventions, provided that it is defined as near.

4. The module definition file of Figure 10-6 lists the total number of words pushed on the stack as parameters when calling the privileged function HercCard. This number is specified within the EXPORTS statement, as follows:

```
EXPORTS
        HERCCARD  @1  2
```

As you have seen, the **@1** assigns the dynamic-link function an ordinal value of 1. The 2 specifies the total number of words contained in the parameters passed to this function. Since HercCard is passed a single double-word parameter (a far pointer), the value 2 must be specified. Even if the I/O privileged routine is not a dynamic-link function, you must provide an EXPORTS statement to specify the number of word

parameters passed to the function. If, however, the function is not passed parameters, an EXPORTS statement is not needed.

5. Finally, the system will not run a process that executes I/O privileged code unless the following command is given in the configuration file:

```
IOPL=YES
```

   Under OS/2 version 1.1, the default state is IOPL=NO; therefore, you must specify IOPL=YES if you want to run privileged code.

The function HercCard reads port number 0x03ba (with the IN instruction). Before reading this port, however, it calls the OS/2 API function **DosPortAccess** (described in Figure 10-9) to request access to this specific port. After it has completed reading the port, it calls DosPortAccess again to relinquish port access. Under OS/2 version 1.1 DosPortAccess does nothing. The function calls are included to provide compatibility with the future 80386-specific version of OS/2 (the 80386 processor can restrict the ports a given process can access). Under OS/2 for the 80386, a process will have to call DosPortAccess to obtain access to a specific port address (or range of port addresses) before an I/O privileged routine, running under this process, can perform a port read or write operation. Thus, even though a routine has I/O privilege, it will not automatically be granted port access.

Under the 80386 version of OS/2, calling DosPortAccess will also automatically grant access to the CLI and STI instructions. If the process uses the CLI or STI instruction, and if it does not call DosPortAccess, it will be required to call the function **DosCLIAccess** (explained in Figure 10-10) to explicitly obtain access to these two instructions. Under OS/2 version 1.1, DosPortAccess and DosCLIAccess can be called from either a privileged or a non-privileged routine within the process that requires access to the privileged instructions. Under version 1.0, however, these functions can be called only from a non-privileged routine. If a frequently called dynamic-link function accesses a port, you could call DosPortAccess once from an initialization routine to request port access and once from a termination routine to release port access, rather than calling DosPortAccess each time the dynamic-link function receives control. (DosCLIAccess could likewise be called from an initialization routine.)

Figure 10-9: The DosPortAccess OS/2 function.

DosPortAccess

☐ *Purpose*: Requests or relinquishes access to a range of I/O ports for the current process (the routine that contains the actual port access instructions must run with I/O privilege). This function also grants access to the CLI and STI instructions.

☐ *Prototype*: USHORT APIENTRY DosPortAccess

(USHORT usReserved,	A reserved value: you must pass a zero value.
USHORT fFunction,	Type of access request: a value of 0x0000 requests access to a range of ports, and a value of 0x0001 relinquishes access to these ports.
USHORT usFirstPort,	First port number in the range of ports.
USHORT usLastPort);	Last port number in the range of ports (to specify a single port, this value should be the same as the usFirstPort parameter).

☐ *Return Value*: If successful, the function returns zero. If an error occurs, it returns a nonzero error code.

Figure 10-10: The DosCLIAccess OS/2 function.

DosCLIAccess

☐ *Purpose*: Grants permission to the current process to use the CLI machine instruction (for disabling hardware interrupts) and the STI machine instruction (for enabling hardware interrupts). The routine that issues either of these instructions must execute with I/O privilege.

☐ *Prototype*: USHORT APIENTRY DosCLIAccess

(VOID);

☐ *Return Value*: If successful, the function returns zero. If an error occurs, it returns a nonzero error code.

As mentioned in Chapter 3, under the protected mode of the 80286 processor, a program thread runs at one of four distinct privilege levels (also known as **rings**, where ring 0 is the highest privilege level, and ring 3 is the lowest). The operating system kernel runs at ring 0 and a normal program thread runs at ring 3. OS/2 is configured so that I/O privilege is granted to a thread running at ring 2 (or at a higher privilege level); thus, under OS/2, an I/O privileged routine is simply one that runs at ring 2. Each code segment is assigned an inherent privilege level. The current privilege level of a given thread is the privilege level of the code segment it is currently executing. Exceptions are **conforming code segments**—a topic beyond the scope of the present discussion.

Also when a program calls a function that runs at a higher privilege level (that is, one contained in a code segment marked with a higher privilege level), the call must pass through a call gate, which is a special segment descriptor that points to the higher privileged function. Furthermore, the 80286 processor requires that code segments executing at different privilege levels must have separate stacks. Thus, an I/O privileged dynamic-link function represents an exception to the rule that a dynamic-link function uses the stack belonging to its client program.

Consequently, when a program running at ring 3 calls an I/O privileged function, the system must copy the function's parameters onto a new stack. Since the system needs to know how many words to copy, you must supply this value in the SEGMENTS statement as seen previously in the chapter. The stack copying operation also demands that the number of parameters be fixed and that the called function restores the stack with the RET *n* instruction; thus, the privileged instruction must conform to the Pascal calling conventions.

Under OS/2 version 1.1, the default size of the stack that the system allocates for an I/O privileged code segment is 512 bytes. As you have seen, however, a routine that calls an OS/2 API function should have a *minimum* stack size of 2 kilobytes. Fortunately, you can increase the default stack size for a privileged routine by calling the OS/2 function **DosR2StackRealloc,** described in Figure 10-11 (this function is not available in OS/2 version 1.0).

DosR2StackRealloc must be called from non-privileged code before the privileged routine is executed (by the same thread that executes the

privileged routine). If the privileged routine is in a dynamic-link library, the library could provide a dynamic-link function in a non-privileged code segment that is called by the client program. This function could call DosR2StackRealloc to adjust the ring 2 stack size for the current thread (if it has not already been adjusted) and then call the actual privileged routine. Thus, the client program would not call the privileged routine directly, but rather would call a non-privileged entry function.

Figure 10-11: The DosR2StackRealloc OS/2 function.

DosR2StackRealloc

☐ *Purpose*: Increases the size of a thread's ring 2 stack (the stack used by an I/O privileged routine executed by the thread) to the specified value.

☐ *Prototype*: USHORT APIENTRY DosR2StackRealloc

(USHORT usSize);	The desired stack size in bytes; the requested value must be larger than the current size (the original default size is 512 bytes).

☐ *Return Value*: If successful, the function returns zero. If an error occurs, it returns a nonzero error code.

DosR2StackRealloc is one of a large number of API functions that cannot be called from an I/O privileged routine. Under OS/2 version 1.0, none of the API functions can be called from I/O privileged code. Under OS/2 version 1.1, however, a subset of the API functions, conforming functions, can be called from I/O privileged code. The functions in this subset are listed in Table 10-2 on the next page.

An I/O privileged routine cannot freely call functions in a non-privileged code segment. The 80286 processor generally does not allow code running at one privilege level to call code running at a lower privilege level). Therefore, if a program or dynamic-link library consists of both privileged and non-privileged code, a non-privileged routine can call a privileged routine. The privileged routine, however, must *return* control to the non-privileged code; it normally cannot directly *call* a non-

privileged function. OS/2 version 1.1, however, provides the function **DosCallback**, which allows a privileged function to call certain routines in non-privileged segments. A function called through DosCallback, however, suffers the severe constraint that it cannot be passed parameters on the stack. See the OS/2 technical documentation for more information on this function.

In general, because of the limited function calling ability of privileged routines, you should limit the amount of code placed in such a routine. The major portion of your code should be in non-privileged segments, and only the sections of code that actually use privileged instructions should be included in a privileged segment.

Finally, since I/O privileged code typically accesses specific port addresses, it tends to be machine-specific. Such code is ideally placed in dynamic-link libraries, where the machine-specific implementation details are hidden from the client application. Also, the flexibility of the dynamic-linking mechanism allows the programmer to easily update the privileged code to support new devices, and to provide a set of alternative routines to match the current hardware configuration.

Table 10-2: API Functions

DosAllocHuge	DosLockSeg
DosAllocSeg	DosMakePipe
DosAllocShrSeg	DosMemAvail
DosBeep	DosMkdir
DosBufReset	DosMove
DosCallback	DosMuxSemWait
DosChdir	DosNewSize
DosChgFilePtr	DosOpen
DosCliAccess	DosOpenSem
DosClose	DosPhysicalDisk
DosCloseSem	DosPortaccess
DosCreateCSAlias	DosQAppType
DosCreateSem	DosQCurDir
DosCreateThread	DosQCurDisk
DosCwait	DosQFHandState

Table 10-2: API Functions

DosDelete	DosQFileInfo
DosDevConfig	DosQFileMode
DosDevIOCtl	DosQFSInfo
DosDupHandle	DosQHandType
DosEnterCritSec	DosQVerify
DosErrClass	DosRead
DosError	DosReadAsync
DosExecPgm	DosReallocHuge
DosExit	DosReallocSeg
DosExitCritSec	DosResumeThread
DosExitList	DosRmdir
DosFileLocks	DosR2StackRealloc
DosFindClose	DosScanEnv
DosFindFirst	DosSearchPath
DosFindNext	DosSelectDisk
DosFlagProcess	DosSemClear
DosFreeModule	DosSemRequest
DosFreeSeg	DosSemSet
DosFSRamSemClear	DosSemSetWait
DosFSRamSemRequest	DosSemWait
DosGetCp	DosSendSignal
DosGetDateTime	DosSetCp
DosGetEnv	DosSetDateTime
DosGetHugeShift	DosSetFHandState
DosGetInfoSeg	DosSetFileInfo
DosGetMachineMode	DosSetVecDosSetFileMode
DosGetModHandle	DosSetVerifyDosSetFSInfo
DosGetModName	DosSizeSegDosSetMaxFH
DosGetPid	DosSleepDosSetPrty
DosGetPPid	DosSubAllocDosSetSigHandler
DosGetProcAddr	DosSubfree
DosGetPrty	DosSubSet
DosGetResource	DosSuspendThread

Table 10-2: API Functions

DosGetSeg	DosTimerAsync
DosGetShrSeg	DosTimerStart
DosGetVersion	DosTimerStop
DosGiveSeg	DosUnlockSeg
DosHoldSignal	DosWrite
DosKillProcess	DosWriteAsync
DosLoadModule	

GLOSSARY

This glossary defines many of the technical terms that have been used in this book, as well as other words you may encounter while working with dynamic-link libraries and OS/2.

abstraction Creation of a function or new data type that can be used without knowledge of the details of the function implementation or data type structure. *See also* **encapsulation**.

alias A segment selector that references the same memory segment as another selector (the corresponding segment descriptors will contain the same physical address, but may contain different access rights).

anchor block handle A numeric value that identifies a process to the Presentation Manager (returned by WinInitialize).

ANSI American National Standards Institute; a body that defines standards for the computer industry. For example, ANSI codes are a standard set of escape sequences that can be embedded in video output to control the console.

API The Application Program Interface; a set of services, implemented as dynamic-link libraries, provided by OS/2 and the Presentation Manager for application programs.

argument A value passed to an operating system command from the command line, or passed to a function within a program; same as **parameter**.

ASCII American Standard Code for Information Interchange; the encoding scheme that represents the character set used by microcomputers. The codes between 0 and 127 are standard among all computer manufactures; in addition, IBM-compatible microcomputers employ the codes between 128 and 255 to represent an extended set of characters.

asynchronous Two or more procedures are said to by asynchronous if they occur concurrently, but without timing relationships. Specifically, if two procedures are asynchronous, when a particular instruction is ex-

ecuted in one procedure it is not possible to predict which instruction is currently being executed in the other procedure. *See also* **synchronous**.

atomic Indivisible; describes a sequence of processor operations that cannot be interrupted, typically those required to execute a single machine instruction.

automatic data segment The physical segment containing the initialized, uninitialized, and constant data defined by a program or dynamic-link library; for an application program, it also contains the program stack and heap.

automatic variable A C program variable declared within the scope of a function without using the static or extern keywords; automatic variables are private to each instance of a function, and are stored within the function's stack frame or machine registers.

background program A program executed through the RUN configuration command, the DETACH command line instruction, or from a parent process. The background program is not attached to a screen group, and does not normally interact with the user.

batch file A file containing a sequence of instructions to be executed by a command interpreter; these files have the .BAT extension in a real-mode environment, and the .CMD extension in a protected-mode environment.

BIOS Basic Input-Output System; the low-level code that controls I/O devices, normally implemented in the read-only memory (ROM) supplied with the computer. Usually, BIOS code can run only in real mode.

bitmap A Presentation Manager data structure contained in a file or in memory that stores a graphic image as a sequence of on or off bits indicating the actual pixel values used to create the image on the screen or other output device. *See also* **metafile**.

block To wait for an event without consuming processor cycles. For example, a thread can block while waiting for a shared resource, and will be released by the operating system when the resource is available.

boot disk The disk drive that contains the code used to initialize the computer and load the operating system when the system is first reset or powered on.

bound program Same as **dual-mode program**.

buffer An area in memory for temporarily storing data that is read or written in blocks; used to increase the efficiency of data transfer operations.

busy waiting Creating a delay, or waiting for a computer event, by executing a non-productive program loop.

cache A high speed storage area used to improve the efficiency of accesses to a slower speed storage medium. For example, temporarily storing data in a disk cache in random access memory can reduce the number of accesses required to transfer contiguous blocks of disk data.

call gate A special type of segment descriptor that allows a process to call a subroutine contained within a higher privileged code segment.

CGA Color Graphics Adapter; a standard video controller that provides graphics with a maximum resolution of 640 by 200 pixels.

child process A process started by another process (its parent).

class A set of attributes, including the address of a window procedure, that defines the behavior of a Presentation Manager window. A class can be registered through the WinRegisterClass function, and every window must be assigned to a class when it is created. A window is known as an **instance** of the class to which it belongs.

click To rapidly press and release a mouse button.

client window The Presentation Manager window created by WinCreateStdWindow that is typically managed by the application program and used to display program data; it is a child of and is owned by the frame window.

client process A process that references a dynamic-link library.

clip To eliminate data written to the screen or other output device that falls outside of a given boundary.

clipboard A Presentation Manager facility for transferring data within a single application or among separate applications.

code page A table used to define a character set for a particular country.

command processor A program that executes operating system commands typed at a prompt. By default, the command processor for protected-mode screen groups is CMD.EXE, and for the real-mode screen group is COMMAND.COM.

compatibility box The OS/2 screen group for running real-mode MS-DOS programs; also known as the **3.x box**.

concurrency The simultaneous execution of two or more sequences of machine instructions. *See also* **multitasking**.

configuration file The file CONFIG.SYS, which contains configuration commands that are read and processed by OS/2 during system initialization.

control window A Presentation Manager window used to receive input or perform a specific function, such as a scroll bar owned by the frame window or a push button owned by a dialog box. Control windows typically send messages to their owners to report user input or other relevant events.

cooked mode A state of a character device driver in which it processes certain characters within the data stream as control codes; also known as the **ASCII mode**. *See also* **raw mode**.

CPU The central processing unit, or the microprocessor belonging to a microcomputer.

critical error A program error due to an external condition at runtime, typically a condition that can be corrected by the user (such as an open door on a disk drive or a printer that is turned off-line).

critical section A body of code accessing a program object (such as a memory variable) that cannot be shared by more than one simultaneous task.

cursor A highlighted area that marks a particular character position on the screen, typically indicating the point at which new characters are inserted into textual data. In some of the literature, this term refers to the mouse pointer.

daemon (1) A supernatural being in Greek mythology that is intermediate between man and the gods; (2) a background program that performs a utility task without direct interaction with the user.

deadlock A situation in which one or more tasks are blocked, waiting for an event that cannot occur.

declaration In C, a construct that specifies the name, data type, and other attributes of a variable or function. *See also* **definition**.

definition In C, a construct that initializes and reserves storage for a variable, or that specifies the name, return type, parameters, and body (the actual code) of a function. *See also* **declaration**.

desktop window The entire Presentation Manager screen, which is the parent of all top-level windows.

descriptor A structure in memory maintained by the operating system, which contains the physical address of a segment as well as other information regarding this segment.

descriptor table A table in memory containing a collection of segment descriptors.

device context Under the Presentation Manager, the physical device associated with a presentation space. *See also* **presentation space.**

device driver A program that translates operating system commands into the device-specific code necessary to control a given device.

DGROUP The name of the group of logical segments that constitute the **automatic data segment.**

dialog box A temporary Presentation Manager window that contains a set of control windows for displaying data and obtaining information from the user.

disjoint descriptor space A range of slots within all local descriptor tables reserved for referencing dynamic-link library segments, as well as shared data segments.

DOS compatibility box Same as **compatibility box.**

double click To click a mouse button twice in rapid succession, while the mouse pointer remains within the same screen area.

drag To press a mouse button and hold the button down while moving the mouse.

dual-mode program A specially prepared program that can run under MS-DOS, in the compatibility box of OS/2, or within a protected-mode screen group; the same as a **bound program.**

dynamic linking A linking method in which the code and data belonging to an external function are stored in a separate disk file and are not

bound to the program until the program is executed. *See also* **static linking.**

dynamic-link library A collection of subroutines stored in a disk file (with the .DLL extension), which may be loaded into memory and called by application programs.

EBCDIC Extended binary coded decimal interchange code; an 8-bit code for character representation, typically used on IBM minicomputers and mainframes.

EGA Enhanced Graphics Adapter; a standard video controller that provides graphics with a maximum resolution of 640 by 350 pixels.

encapsulation Refers to hiding the details of the implementation of a function or data structure. According to the ideal of code and data encapsulation, the application programmer is free to call the function or use the data structure, but does not have access to the inner details of the code or data. *See also* **abstraction.**

entry point name The name of a dynamic-link function as it is listed in the header of the dynamic-link library (and also within the header of the client program, unless it is referenced by **ordinal value**). *See also* **external name.**

entry table A list within the header of a dynamic-link library file containing the entry point address of each dynamic-link function defined in the library; this table is indexed by the **ordinal values** of the functions.

exception A processor error (such as an attempt to divide by 0) or other internal processor condition, which generates an interrupt and transfers control to a software routine designed to handle the event.

expanded memory Memory available to real-mode programs above the normal 640-kilobyte limit. This memory is contained on special adapter cards and is accessed through a hardware paging mechanism.

export To list a dynamic-link function under the EXPORTS statement of the module definition file so that it can be called by a client program.

Extended Edition A version of OS/2 that contains operating system extensions developed by IBM, such as a data base manager or a communications manager.

extended memory Memory contained in 80286/80386 machines at addresses above 1 megabyte. This memory space can be directly accessed only in protected mode.

external function In a C program, a function declared within the current source file that is defined within another source module; it can be statically or dynamically linked to the current module.

external name The name of a dynamic-link function used within the client program source code.

external reference A record within an object module of the name of a code or data item reference by the module but defined within another module.

external variable A C program variable declared outside the scope of a function.

family API The OS/2 API functions that can be called by dual-mode programs when they are running in real mode; the family API functions form a subset of the full API.

far address An address containing both a 16-bit segment **selector** (in protected mode, or a segment **address** in real mode) and a 16-bit **offset** from the beginning of this segment. *See also* **near address**.

fast-safe semaphore A special type of semaphore that allows a dynamic-link library to synchronize the activities of multiple clients; this form of semaphore offers the speed of a RAM semaphore, but can be

cleared by a termination routine if the process terminates abnormally. *See also* **semaphore**, **RAM semaphore**, and **system semaphore.**

file handle A number used to identify a file opened under OS/2.

focus window The Presentation Manager window to which the system sends all keyboard messages.

frame window The top-level Presentation Manager window created by WinCreateStdWindow, which owns and is the parent of the other windows generated by this function.

gigabyte 2^{30}, or approximately one billion, bytes.

global data segment A data segment defined by a dynamic-link module that is shared by all client processes. *See also* **instance data segment.**

global descriptor table A table in memory containing segment descriptors that can be accessed by all processes in the system. *See also* **local descriptor table.**

global initialization Execution of a dynamic-link library initialization routine only when the module is referenced by the first client process. *See also* **instance initialization.**

handle A numeric value returned by many OS/2 and Presentation Manager functions used to identify the owner of a program object (such as an open file or a Presentation Manager window) when subsequent function calls are made.

hardware interrupt A processor event triggered by an external device (such as the keyboard) or by the processor itself, which temporarily suspends the execution of the current process and causes branching to a routine that provides an appropriate service.

header file Same as **include file.**

heap A block of memory out of which smaller blocks of memory are dynamically allocated. In C, functions such as malloc allocate blocks of memory from the C program heap.

Hercules Graphics Card (HGC) A video adapter (produced by Hercules Computer Technology) that supports the standard monochrome text mode, as well as a high-resolution monochrome graphics mode (720 by 348 pixels).

hexadecimal A base-16 number system.

hotkey A keystroke that activates a background program (usually a monitor), switches screen groups, or invokes the Task Manager.

hot spot A single, specially designated pixel within a mouse pointer; the position of the mouse pointer is specified as the coordinates of the hot spot.

huge memory allocation A block of allocated memory consisting of more than one segment. The segments may not be contiguous in memory, but the selectors for these segments differ numerically by a constant amount.

icon A fixed-size graphic image that you can create and display within a Presentation Manager application; also, an application window is represented by an icon when it is minimized.

import library A library file that resolves references to dynamic-link functions by supplying records that contain the module name and function entry point, but not the actual code.

include file A file that the preprocessor merges into a C program file in response to the #include directive.

indirection In the C language, accessing the value of a variable or calling a function through a pointer that contains the address of this variable or function.

instance data segment A data segment defined by a dynamic-link library that is not shared by multiple client processes; the system loads a separate copy of an instance data segment for each new client.

instance initialization Execution of a dynamic-link library initialization routine each time it is referenced by a new client process. *See also* **global initialization**.

interprocess communication Sending signals or exchanging data among separate threads or processes.

interrupt A software, hardware, or processor generated event that passes control to a routine in memory, which provides a service or handles a condition.

I/O privilege Permission granted to a specific code segment to issue direct port I/O instructions or to enable or disable hardware interrupts.

kernel The core operating system code that operates at the highest privilege level; also, the portion of the API exclusive of the Presentation Manager and other operating system extensions.

kernel application A protected-mode program that calls only the basic system services (the Dos, Kbd, Mou, and Vio functions); it does not use the special services of the Presentation Manager or other operating system extensions.

kilobyte 2^{10}, or 1024, bytes.

library file (.LIB file) A file that stores one or more object modules; it is created and maintained by the Microsoft LIB utility. The LINK program can extract referenced object modules directly from a library file. It is not the same as a **dynamic-link library** (.DLL file).

loadtime dynamic linking A form of linking in which all dynamic-link modules referenced by an executable file (a program or dynamic-link

library) are automatically read into memory when the file is loaded. *See also* **runtime dynamic linking**.

local descriptor table A table in memory containing descriptors that can be accessed by a specific process. *See also* **global descriptor table**.

logical segment A term used in this book to refer to a segment defined by the compiler or assembler; if several logical segments are placed in a segment group, the linker generates only a single physical segment containing these logical segments. *See also* **physical segment**.

machine instruction A single binary instruction processed directly by the CPU. In assembly language, a machine instruction is represented by mnemonic symbol, such as MOV or JMP. A high-level language instruction ultimately generates one or more machine instructions.

MAKE A utility for preparing a program, which reads a script from a text file (known as a **MAKE file**) and performs only those steps necessary to build an updated version of the program.

maximize To enlarge a Presentation Manager window to its maximum size, which fills most of the screen.

maximize box A Presentation Manager control window displaying an upward pointing arrow; when the user clicks the mouse with the pointer within this control, the owner (or parent) window is maximized.

MDA Monochrome Display Adapter; a video adapter that supports the standard monochrome text mode (80 columns by 25 lines).

megabyte 2^{20}, or approximately 1 million, bytes.

memory model For a C program, the set of default address types (near or far) used for variables, pointers, and functions, and the manner in which segments are arranged in memory. The memory model can be small, medium, compact, large, or huge.

menu A Presentation Manager control window used to display a list of commands; the user issues a command by selecting the associated item from the menu.

message box A temporary Presentation Manager window that displays a message and pauses for user input.

message queue A data structure associated with a Presentation Manager program thread, which is used to store messages posted to any of the windows created by that thread.

metafile A Presentation Manager data structure contained in a disk file or in memory that stores a graphic image as the sequence of commands required to create the image. *See also* **bitmap**.

minimize To reduce a Presentation Manager window to an icon on the screen.

minimize box A Presentation Manager control window displaying a downward pointing arrow; when the user clicks the mouse with the pointer within this control, the owner (or parent) window is minimized.

module A term referring either to the object code generated from a single source file (an .OBJ file or the contents of such a file within a .LIB file), or to a dynamic-link library (a .DLL file).

module definition file A text file (with the .DEF extension) used to specify a wide variety of features when generating a program or dynamic-link library with the linker.

monitor An OS/2 program that processes the stream of characters passing to or from a character device.

mouse An input device that the user moves on the desk surface, causing a pointer to move on the screen; a mouse has 1 to 3 buttons that transmit information to the program.

multiprocessing Simultaneous execution of code by more than one processor in a single computer.

multitasking Simultaneous execution of more than one sequence of machine instructions.

multiuser Refers to a single computer connected to more than one terminal, which allows several users to share the same processor. OS/2 is not a multiuser system.

name table A list within the header of a dynamic-link library file containing the name and ordinal value of each dynamic-link function defined in the file.

near address An address consisting of only a 16-bit offset. *See also* **far address**.

nonshared data segment Same as an **instance data segment**.

object file A file containing the binary machine instructions and data generated by the C compiler or the assembler from a single source file.

object window A Presentation Manager window used to receive messages; an object window is not displayed on the screen, does not receive input messages, and does not have a parent.

offset A 16-bit displacement that must be combined with a 16-bit segment value to access a memory location under the Intel microprocessor architecture.

ordinal value A number that identifies the entry point of a function within a dynamic-link library; it is an index into the **entry table** within the header of the dynamic-link library.

over-commitment (of memory) Allocation of more memory than is physically present in the machine; accomplished by swapping segments between memory and secondary storage on a disk.

parameter A value passed to an operating system command from the command line, or passed to a function within a program; same as **argument**.

parent process A process that starts another process (its child).

parent window A Presentation Manager window within which one or more child windows are displayed.

pascal conventions The naming and calling conventions used for a function declared with the pascal keyword; namely, (1) parameters are pushed on the stack in the same order that they are listed in the function prototype; (2) the function removes the parameters from the stack; (3) the function has a fixed number of parameters; (4) the function name is converted to uppercase before being written to the object file.

path The complete specification of the location and name of a disk file, including the drive, directory, and file name.

physical segment A term used in this book to refer to a code or data segment allocated by the system and accessed through a segment selector; a single physical segment may consist of several logical segments defined by the compiler or assembler. *See also* **logical segment**.

pipe A form of interprocess communication under OS/2 that allows two related processes to exchange a serial stream of data.

pixel Picture element; smallest unit on the screen that can be controlled (turned on or off, or assigned a color or intensity).

point To place the hot spot of the mouse pointer on a given item or within a given area.

pointer (1) A program variable that contains the address of another variable or of a function; (2) a fixed-size graphic image that is moved on the screen in response to movements of the mouse, or in response to program function calls.

poll To test for the occurrence of a specific event; for example, an inefficient program might repeatedly poll for the arrival of a character rather than blocking.

port An address used to communicate with a peripheral device.

post To place a message in a Presentation Manager message queue.

pragma A statement within a C program that controls the operation of the compiler.

preempt To take control away from a particular task in a multitasking system (rather than letting the task voluntarily yield control). OS/2 is a preemptive multitasking system.

Presentation Manager An operating system extension provided with OS/2 versions 1.1 and later, which provides a windowed, graphics interface similar to Microsoft Windows.

Presentation Manager application A program written specifically for the Presentation Manager, which performs the necessary initializations and calls Presentation Manager API functions.

presentation space The abstract space onto which a Presentation Manager application displays data; it must be associated with a physical device (known as the **device context**). A presentation space is maintained as a data structure by the system, and is assigned device-independent attributes, such as a current set of colors and a current font.

priority A value assigned to each thread in the system that allows the scheduler to determine which thread among those ready to run should be dispatched.

private function A term used in this book to refer to a function within a dynamic-link library that can be called only by other functions within the library, and cannot be called by client program. *See also* **public function**.

privilege level The set of permissions associated with a particular segment. There are four privilege levels, numbered from 0 (the highest level, reserved for the operating system) to 3 (the lowest level, for application programs). A given privilege level is also known as a **ring** (such as *ring 3*).

process A single instance of the execution of a program; under OS/2, a process is also the unit of ownership of objects such as memory segments and file handles.

program A collection of code and data stored in an executable file and loaded into memory at runtime.

protect To prevent a given process from corrupting other processes in a multitasking system; OS/2 uses the hardware protection mechanisms provided by the 80286 processor.

protected mode A processor state of the 80286/80386 processors that allows the operating system to safely run multiple tasks and provide virtual memory; same as **virtual mode**. *See also* **real mode**.

protection fault A processor exception generated when an application attempts to violate the protection mechanisms enforced in the protected mode. *See also* **exception**.

prototype In C, a function declaration that includes a list of the names and types of the parameters.

public function A term used in this book to refer to a function within a dynamic-link library that is exported and can therefore be called by a client program. *See also* **private function**.

queue A form of interprocess communication provided by OS/2; specifically, an ordered collection of messages that have been sent from one or more processes to a single receiving process (not related to a Presentation Manager **message queue**).

RAM Random access memory; the randomly addressable, volatile main memory used to store code and data.

RAM semaphore A semaphore stored in a variable defined by a program; this type of semaphore is fast, but is suitable for use only within a single process. See also **semaphore**, **fast-safe semaphore**, and **system semaphore**.

raw mode A state of a character device driver in which it passes all characters as literal values, and does not respond to control codes embedded in the data stream; also known as **binary mode**. *See also* **cooked mode**.

real mode A processor state of the 80286/80386 processors that emulates the operation of the 8086/8088 processors. *See also* **protected mode**.

real-mode screen group Same as **compatibility box**.

reentrant A body of code is reentrant if it can be executed more than once at a given time.

registers Locations used to store values within the processor; each register has a label, such as AX and CS, and is used for a specific set of purposes.

relocation A process performed when loading a program or dynamic-link library, in which the actual addresses are written to all unresolved address references within the program segments; the addresses may be of code or data items within the current executable file or within a dynamic-link library.

relocation record A record that allows the loader to supply the address of a given object to all references to this object within a segment. Each segment within a program or dynamic-link library has its own table of relocation records; a given relocation record identifies the addressed object (a code or data item within the current executable file, or a

dynamic-link function) and contains a pointer to a list of all references to this object within the segment.

resource A form of read-only data stored within a special segment in a program file or dynamic-link library. This data is inserted directly into the file by a resource compiler, and is loaded into memory when required by an application.

resource compiler An OS/2 utility that translates a text script into binary resource data, and inserts the data directly into a program file or dynamic-link library.

resource script A text file that defines a set of OS/2 resources.

ring Same as **privilege level**.

ROM Read only memory; non-volatile memory supplied with microcomputers that normally contains code for controlling hardware devices.

runtime dynamic linking A form of linking in which dynamic-link libraries are explicitly loaded by a program through the DosLoadModule system service. *See also* **loadtime dynamic linking**.

runtime library A collection of functions that can be called by a program, typically supplied by a compiler.

scan code A hardware-specific code emitted by the keyboard to indicate the key that has been pressed or released.

scheduler The portion of the operating system that apportions CPU time among multiple threads, and determines the priority of each thread.

screen group A collection of processes that share a single virtual screen, keyboard, and mouse; same as **session**.

scroll To move a block of data displayed within a Presentation Manager window toward one of the four window borders.

scroll bar A Presentation Manager control window used to scroll textual data within a window. A horizontal scroll bar is typically located along the lower edge of its parent and scrolls text horizontally; a vertical scroll bar is typically located along the right edge of its parent and scrolls text vertically.

segment A variable length block of allocated memory (under the 80286 processor, its maximum length is 64 kilobytes); in protected mode, each segment has an associated set of access rights.

segment descriptor Same as **descriptor**.

selector The virtual address of a memory segment, which serves as an index into a table containing the actual physical address (the **descriptor table**); to address a segment, the corresponding selector is loaded into a segment register such as DS.

semaphore A software flag used to synchronize the activities of two or more threads of execution. See also **fast-safe semaphore**, **RAM semaphore**, and **system semaphore**.

serialize To assure that a given object can be accessed by only one thread at a time; access to an object is typically serialized by means of a semaphore.

server The process that owns an OS/2 queue and receives the queue messages.

session Same as a **screen group**.

session manager A system utility that manages switching among screen groups; in OS/2 version 1.1, this task is performed by the Presentation Manager.

shared data segment Same as a **global data segment**.

sizing border A Presentation Manager control window that forms a wide border around its parent window; it can be used to adjust the size of the parent window with the mouse.

spooler A background program that stores printer output in temporary files and prints these files in a given order.

stack A data structure in memory that holds the parameters and local variables belonging to a function while the function is active. It is also used for the temporary storage of register contents or memory variables.

stack frame The area of the stack containing the local variables, return address, and parameters associated with a given function call.

static A C storage class; for a variable declared within the body of a function, it causes the variable to retain its value between function calls; for a variable or function declared outside a function, it renders the object accessible within the portion of the current file following the declaration, but not within other files.

static linking The conventional linking method, in which the code and data belonging an external function are physically bound to the program file by the linker. *See also* **dynamic linking**.

string In C, an array of characters terminated with a NULL character ('\0').

stub loader The routine within a dual-mode program that receives initial control if the program is loaded under real mode; it loads the real-mode versions of all dynamic-link functions and prepares the executable image to run under real mode.

stub program A routine within a protected-mode program that receives control if the user attempts to load the program under real mode; it typically prints an error message and terminates the process.

subsystem A set of dynamic-link functions that control a shared device or other object, such as the OS/2 video management services (the *Vio* functions).

swapping The temporary storage of memory segments on disk to make room for other segments. This mechanism is useful when memory is **overcommitted**.

synchronous Two procedures are said to be synchronous if their actions occur in a specific order. For example, if one procedure waits at a given point until a second procedure completes, these procedures are said to be synchronous. *See also* **asynchronous**.

system semaphore A semaphore managed by the system that can be accessed (through its name) by multiple processes; if a process terminates while holding a system semaphore, it is automatically cleared. *See also* **semaphore**, **fast-safe semaphore**, and **RAM semaphore**.

task A general term referring to one of the concurrent threads or processes of a multitasking system.

Task Manager A Presentation Manager utility that allows the user to switch to a selected program, to terminate a process, or to shut down the system.

thread The basic dispatchable entity under OS/2; the execution of a series of machine instructions within a program.

time slice The period of time that the scheduler allows a thread to run before it grants CPU time to another thread of equal priority.

timeout To exhaust the designated waiting period for a given event; for example, if the system is unable to write to the printer, it will generally wait for a specified timeout period before returning an error code.

title bar A Presentation Manager control window typically located along the top edge of its parent; the title bar contains a text title and is used for moving the parent window with the mouse.

top-level window A Presentation Manager window that is the direct child of the desktop window; also known as a **main window**.

virtual memory The allocatable memory space in protected mode, which may exceed the amount of physical memory installed in the machine. Segments within this space may be temporarily swapped to a disk to make room for segments that are currently being accessed.

virtual mode Same as **protected mode**.

VGA Video Graphics Array; a standard video controller that normally provides a maximum graphics resolution of 640 by 480 pixels.

window The fundamental object owned by a Presentation Manager application, which can receive messages and is usually associated with a rectangular area on the screen used to interact with the user.

window class *See also* **class**.

window procedure The function within a Presentation Manager application that processes the messages sent to a given window (when a message is sent, the system calls this procedure, passing the content of the message as parameters).

word A two-byte data value; under Intel processors it is stored in memory with the low-order byte first (that is, at the lower address), followed by the high-order byte.

BIBLIOGRAPHY

Comer, D. *Operating System Design, the XINU Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

Cortesi, D. *The Programmer's Essential OS/2 Handbook*. Redwood City, CA: M & T Books, 1988.

Duncan, R. *Advanced OS/2 Programming*. Redmond, WA: Microsoft Press, 1989.

Harbison, S. and Steele, G. *C: A Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

IBM. *IBM OS/2 Technical Reference Version 1.0* (part number 6280201). IBM Technical Directory, P.O. Box 2009, Racine, WI 53404-3336: IBM Corporation, 1988.

Intel. *80286 and 80287 Programmer's Reference Manual*. Santa Clara, CA: Intel, 1987.

Intel. *80286 Operating Systems Writer's Guide*. Santa Clara, CA: Intel, 1986.

Kernighan, B. and Ritchie, D. *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Lafore, R. *Assembly Language Primer for the IBM PC & XT*. New York, NY: New American Library, 1984.

Letwin, G. *Inside OS/2*. Redmond, WA: Microsoft Press, 1988.

Microsoft. *Mixed-Language Programming Guide*. (supplied with high-level language compilers) Redmond, WA: Microsoft Corporation, 1987.

Microsoft. *Operating System/2 Programmer's Toolkit, Programmer's Learning Guide and Programming Tools*. Redmond, WA: Microsoft Corporation, 1988.

Microsoft. *Operating System/2 Programmer's Toolkit, Programmer's Reference*. Redmond, WA: Microsoft Corporation, 1988.

Microsoft. *The Microsoft OS/2 Programmer's Reference Library* (Volume 1, 2, and 3). Redmond, WA: Microsoft Corporation, 1989.

Petzold, C. *Programming the OS/2 Presentation Manager*. Redmond, WA: Microsoft Press, 1989.

Strauss, E. *Inside the 80286*. New York, NY: Prentice-Hall, 1986.

Young, Michael J. *MS-DOS Advanced Programming*. Alameda, CA: SYBEX, 1988.

Young, Michael J. *Programmer's Guide to OS/2*. Alameda, CA: SYBEX, 1988.

Young, Michael J. *Programmer's Guide to the OS/2 Presentation Manager*. Alameda, CA: SYBEX, 1989.

INDEX

.286 directive, 284

A

abstraction (of code and data), 89
_acrtused, 111, 212, 222
anchor block handle, 49, 55
API (application program interface), 5
  categories of, 6
    using, 6-14
API.LIB, 252-258, 269-270
assembly language, 181, 277-305
    for privileged routines, 291-305
    guidelines, 278-291
/ASu (compiler flag), 132, 223
/ASw (compiler flag), 19, 21, 131, 223, 273
automatic data, 112
automatic data segment, 147-148

B

_beginthread (C library function), 204-208, 223
BIND utility, 252-258, 270
BIOS, 13
bound applications. *See* dual-mode programs

C

/c (compiler flag), 15
C runtime libraries, 5, 16, 199-225
    and dual-mode programs, 270
    for multiple-thread applications, 200-208
    for multiple-thread DLLs and applications, 213-223
    for single-thread DLLs, 208-213
    summary of, 224
    versions of, 199-200
    See also library files
call gate, 87, 300
CALL instructions, 70, 73, 75-79, 81-82, 87
class. *See* window class
client window, 55, 58
.CODE directive, 285
CODE statement (module definition file), 299
coding, guidelines, 109, 111-122
    compiling, 14-15, 19-21, 28, 70-71
    flags for, 15, 19, 21
CRTLIB.DLL, 213-223

D

data categories (in C), 112
.DATA directive, 285
DATA statement (module definition file), 127, 146-149, 290
declarations, 9-10, 122
descriptor table, 150-154
device sharing, 115
DGROUP, 289
disjoint descriptor space, 236-240
Dos functions, 6-7
DOSCALLS.LIB. *See* OS2.LIB

DosCLIAccess (OS/2 function), 301-302
DosCreateThread (OS/2 function), 24-25, 204-206
DosDevIOCtl (OS/2 function), 13, 120-121
DosExit (OS/2 function), 9, 204
DosExitList (OS/2 function), 194-196
DosFreeModule (OS/2 function), 234-235
DosGetInfoSeg (OS/2 function), 257
DosGetMachineMode (OS/2 function), 256-257
DosGetProcAddr (OS/2 function), 230-233
DosLoadModule (OS/2 function), 228-230
DosOpen (OS/2 function), 120, 286-287
DosPortAccess (OS/2 function), 301-302
DosR2StackRealloc (OS/2 function), 303-304
DosSemClear (OS/2 function), 33-35
DosSemRequest (OS/2), 33-34
DosSetSigHandler (OS/2 function), 197-198
DosSleep (OS/2 function), 7, 36
DosWrite (OS/2 function), 121
DS register, 21, 131-132, 289
dual-mode programs, 252-258
dynamic-link object record, 72, 74

E

END statement, 176, 181, 212
_endthread (C library function), 204-208, 223
entry point (of program), 176-177, 181, 183
error handling, 12, 287
    for Presentation Manager, 54
exit list. *See* termination routines
EXPORTS keyword (module definition file), 65-66, 127-128, 130-131, 290, 300
Extended Edition (of OS/2), 88
extensions (to OS/2), 88
external functions, 2, 5-6, 70-71
EXTRN keyword, 183, 290

F

family API, 256-257, 269-270
far (C keyword), 116-117
fonts, 39
frame window, 56

G

global data segments, 80, 87, 112, 148-154
/G2 (compiler flag), 15, 273
global descriptor table (GDT), 152-174
Gpi functions, 50

/Gs (compiler flag), 27, 132
/Gw (compiler flag), 66

H

handle, printer, 120
header files, OS/2, 9-12, 65, 96, 122-124, 207
  for assembly language, 288
HEAPSIZE statement (module definition file), 65
Hercules graphics card, 291-298

I

IMPLIB utility, 97, 130, 133
import library. *See* library files
import record, 17
IMPORTS statement (module definition file), 71-72, 74-75, 97, 138-139
include files. *See* header files
initialization of Presentation Manager, 49
initialization routines (for dynamic-link libraries), 81, 176-187, 212, 230
instance data segments, 80, 87, 112, 127, 143-148
interprocess communication, 28-36. *See also* semaphores
interrupts, 5
I/O privileged routines, 13, 291-305

K

KbdCharIn (OS/2 function), 8
KBDKEYINFO (structure), 11

L

LIB environment variable, 17
LIBPATH configuration command, 17, 78
library files, static, 71
    import, 16-17, 71-72, 74, 97, 133, 204
    See also C runtime libraries
LIBRARY statement (module definition file), 96, 126-127, 184-186, 290
linking, static, 14-21, 28,
    vs. dynamic, 71-77
LLIBCDLL.LIB, 208-213
LLIBCMT.LIB, 200-208
load on call segments, 81
loading programs, 78-83
loadtime dynamic linking, 70, 227
local descriptor table (LDT), 152, 237-240
_loadds (C keyword), 117

M

MAKE utility, 66-67
macro definitions, 12
menus (Presentation Manager), 56-57, 63-65
memory accesses, 13
memory models, 147
message queue (Presentation Manager), 55, 58-60
messages (Presentation Manager), 55, 58-60
.MODEL directive, 284, 289
module, 96

module definition files (.DEF), 17-19, 65-66, 125-131, 138-139, 290
multitasking, 19-28, 111-112, 215
/MX flag, 186, 290-291

N

NAME statement (module definition file), 18-19, 65, 96, 126-127
/NOI (linker flag), 15-16
/NOD (linker flag), 16, 204
nonshared data segment. *See* instance data segment

O

object modules, 71-73
ordinal values, 81-82, 128-131, 140, 232-233
OS2.H (header file), 9-10, 17
OS2.LIB, 16-17, 71-72, 97, 204

P

pascal (C keyword), 116
Pascal conventions, 66, 300
pointers to functions, 233-234, 247-249
ports, 13, 298, 301-302
preload segments, 81
Presentation Manager,
    and dynamic linking, 38-39
    architecture, 40
    programs for, 37-67
presentation space, 62-63
printing, 115, 120-122
private functions, 115
privilege levels, 86-87, 303. *See also* I/O privileged routines
process, 27-28
protected-mode programs, 1-36
    restrictions on, 13
PROTMODE keyword (module definition file), 19
PUBLIC directive, 288, 290-291
public functions, 115

Q

queue, message. *See* message queue (Presentation Manager)

R

real-mode memory addressing, 151
real-mode version of DLL, 251-275. *See also* dual-mode programs
    guidelines for, 269-271
    writing of a, 258-271
reentrant vs. nonreentrant code, 27, 204
registers, saving, 118, 288-289
relocation, 237
relocation record, 75-76, 78, 81-82, 130
resident functions, 87
resource compiler, 66-67
resources (OS/2 and Presentation Manager), 38-39, 64-65, 66-67
returning values (from assembly language routines), 290
rings. *See* privilege levels

runtime dynamic linking, 70, 227-249
advantages of, 235-236
basic steps for, 228-235
example of, 240-249

S
_saveregs (keyword), 118
screen groups, 28, 147
segment descriptors, 150-154
SEGMENTS statement (module definition file), 299-300
selectors, 150
semaphores, 28-36, 114, 172, 213
sessions, 28, 147
shared data segment. *See* global data segment
sharing of data, 113-114, 141-174. *See also* global data segments, instance data segments
SS register, 21, 131-132
stack, 21, 131-132
in assembly language, 289
for dynamic-link libraries, 147-148
for new threads, 25-27, 206
STACKSIZE keyword (module definition file), 65
standard window, 55-58
startup code, 111
structure definitions, 11
stub loader, 252, 255-256
stub program, 255
subsystems, 9, 87
symbolic constants, 11-12, 122

T
termination routines, 85, 187-198
threads, 19-20
time slice, 27
type definitions, 11

U
unresolved references, 72

V
VioWrtTTy (OS/2 function), 10
virtual memory, 150-154

W
/W2 (compiler flag), 15
windows
creating, 55-58
window class, 55, 59
window procedure, 55, 59-62
WINDOWAPI keyword (module definition file), 65
WINDOWCOMPAT keyword (module definition file), 18-19
Win functions, 50
WinBeginPaint (Presentation Manager function), 50, 62
WinCreateMsgQueue (Presentation Manager function), 40, 50
WinCreateStdWindow (Presentation Manager function), 40, 50
WinDefWindowProc (Presentation Manager function), 51, 62
WinDestroyMsgQueue (Presentation Manager function), 51-52, 64

WinDestroyWindow (Presentation Manager function), 52, 64
WinDispatchMsg (Presentation Manager function), 52
WinDrawText (Presentation Manager function), 52, 62
WinEndPaint (Presentation Manager function), 52, 63
WinGetLastError (Presentation Manager function), 54
WinGetMsg (Presentation Manager function), 53
WinInitialize (Presentation Manager function), 40, 49, 53
WinMessageBox (Presentation Manager function), 53
WinQueryWindowRect (Presentation Manager function), 53-54, 62
WinRegisterClass (Presentation Manager function), 40, 54-55
WinTerminate (Presentation Manager function), 54, 64
WM_COMMAND (Presentation Manager message), 61, 63, 65
WM_PAINT (Presentation Manager message), 61-62

Z

/Zp (compiler flag), 15

Unique Programmer's Tools for MS-DOS and OS/2
Developed by Michael J. Young

I am proud to offer the following high quality software tools for MS-DOS and OS/2. These tools represent the culmination of my efforts in developing many C applications and writing seven advanced C programming books; all have proven their usefulness and reliability. They greatly facilitate writing programs and serve as invaluable tools for learning high-performance programming techniques and advanced features of MS-DOS and OS/2.

☐ Systems Tools for C (MS-DOS 2.0 and later)

Systems Tools for C is a comprehensive library of C functions providing virtually all the tools you need to develop high-performance C programs, as well as memory resident utilities. It includes a complete set of functions for managing screens and windows, for controlling the keyboard and printer, for writing critical-error and interrupt handlers, for accessing expanded memory, for creating a mouse interface, and for converting a C program into a TSR activated with a hotkey. It also includes utility, file management, and graphics functions, as well as an interactive screen designing program. The package provides complete commented source code and supports the Borland Turbo C and the Microsoft Optimizing and QuickC compilers. No royalties are required.

☐ PM/Edit (OS/2 1.1 and later)

Now you can write programs and other text files within a window of the Presentation Manager. *PM/Edit* is a full-featured text editor written specifically for the Presentation Manager. It provides the following features: an online help facility; cut, paste, and other block operations; search and replace commands; running a compiler or printing a file while editing; macros; an undo command; word processing features; and more. Complete commented source code is available to allow you to fully customize this editor.

☐ OS/Tools (OS/2 1.0 and later)

OS/Tools is a set of software tools designed to give you a head start in writing OS/2 kernel and Presentation Manager applications. It includes a comprehensive collection of dynamic-link library functions, as well as a set of programmer's utilities. The function library is carefully tailored to the needs of the OS/2 and PM programmer, and serves to extend the OS/2 API and the standard C library. The utilities are designed to exploit unique and interesting features of OS/2; they include programs for designing text mode screens and windows, for writing and executing keyboard macros, for copying data between screen groups, and for capturing and printing data from OS/2 screens. All functions and utilities are written in C and are provided with complete commented source code.

Complete Documentation. Each of these three software products includes a complete manual on disk, plus a convenient program for finding topics and reading the documentation. If you wish, you can also print the manual on any printer.

Introductory Prices. Each of these products is being offered for the introductory price of $49.50 (PM/Edit with source code is $99.50).

Ninety Day Trial Period. You can return any product, for any reason, within 90 days of the date you receive it, for a full refund.

Free Brochure. Order any of these products now, or write or call for a free brochure describing these and other unique programmer's tools and books for MS-DOS and OS/2.

Copies of Systems Tools for C @ $49.50 each _____
Copies of PM/Edit @ $49.50 each _____
Copies of PM/Edit with source code @ $99.50 each _____
Copies of OS/Tools (version 1.1) @ $49.50 each _____
California residents: Add 6% sales tax _____
Shipping and Handling: Add $2.50 ($5.00 for UPS COD or foreign orders) _____
Total Order _____

☐ 5-1/4" diskettes ☐ 3-1/2" diskettes

☐ Please send me a brochure.

Name_____
Address _____
City/State/Zip _____

Please send a check for full payment or request UPS COD (no purchase orders or bank cards; for foreign orders, please send an international money order in U.S. dollars). Your software will be shipped immediately. Order from:

Young Software
20 Sunnyside Avenue, Suite A
Mill Valley, CA 94941
415/383-5354

AW01

IBM Programming/OS/2

A BENCHMARK BOOK

> $22.95 FPT USA

SOFTWARE TOOLS FOR OS/2®

Creating Dynamic Link Libraries

Michael J. Young

Dynamic link libraries (DLLs) are one of OS/2®'s most powerful features. DLLs are the function libraries stored in a separate disk file that are bound to the application at load time or execution. They allow you to incorporate library modules directly into applications, freeing you from the task of writing and rewriting windowing, I/O, and file handling functions. By using OS/2's dynamic link libraries, and by building your own DLLs, the true power and versatility of OS/2 will be at your command.

Software Tools for OS/2: Creating Dynamic Link Libraries tells you how DLLs work, how to use them, and how to create your own DLLs. After a concise review of OS/2 and Presentation Manager programming techniques, Michael Young presents a detailed discussion of how DLLs actually work in the context of compiling, linking, loading, and terminating a program. Once the techniques for writing a simple DLL are thoroughly investigated, the remaining chapters provide detailed, hands-on coverage of DLL capabilities including:

- how to define both shared and non-shared data segments so that a DLL can either share data among all programs or provide data privately to each program
- how to create DLL initialization and termination routines. These routines are especially valuable for DLLs that manage resources shared by several programs
- how special versions of the C runtime library support multiple thread applications and DLLs
- how a program can explicitly load a selected DLL at runtime
- how to provide a real-mode version of your DLL to be used by programs designed to run under real or protected mode.

All examples are written in C, and Chapter 10 describes methods for writing DLLs in assembly language.

Software Tools for OS/2: Creating Dynamic Link Libraries provides an in-depth guide to utilizing one of OS/2's most impressive and important features.

Michael J. Young is a software engineering consultant, a developer of programmer's tools, and an experienced author. His previous books include *MS-DOS Advanced Programming* and *Programmer's Guide to OS/2*.

Cover design by Doliber Skeffington

Addison-Wesley Publishing Company, Inc.

52295

9 780201 517873

ISBN 0-201-51787-6