Osborne McGraw-Hill

OS/2™ PROGRAMMING:

AN INTRODUCTION

Get Up to Speed Fast

Herbert Schildt

OS/2™ PROGRAMMING: AN INTRODUCTION

Herbert Schildt

Osborne **McGraw-Hill**
Berkeley, California

Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to Osborne **McGraw-Hill** at the above address.

A complete list of trademarks appears on page 371.

OS/2™ PROGRAMMING: AN INTRODUCTION

1234567890 DODO 898

ISBN 0-07-881427-8

CONTENTS

Appendixes

ACKNOWLEDGMENTS

Quite a few OS/2 macro names, type names, structure names, and union names have been presented and discussed in this book. These names originate from the OS/2 Developer's Toolkit manuals and disk files and are used with permission of Microsoft Corporation.

Special thanks to William H. Murray III and Chris H. Pappas for allowing me to adapt two assembly language programs in their book *Assembly Language Programming Under OS/2* (Osborne/McGraw-Hill, 1989).

—*H.S.*

PREFACE

The purpose of this book is to give you a "jump start" into the world of OS/2 programming. OS/2 is a big program, and the ways that you, the programmer, can interact with it are numerous and varied. This book will help you understand quickly the essence of OS/2.

The impact of OS/2 is not to be underestimated. OS/2 has done for microcomputer operating systems what the original IBM PC did for microcomputers: in one bold stroke it has defined a new, more powerful computing environment. What makes OS/2 so exciting is that it is the first multitasking operating system designed specifically for the personal, single-user computer. Although microcomputers have been able to run multitasking operating systems such as UNIX for several years, the results have never been entirely satisfactory, partly because the porting of a multi-user, multitasking operating system to a single-user, highly interactive environment generally produced the worst of both worlds: slow response time combined with an old, TTY-based interface. OS/2 maintains the highly interactive nature of the personal computing environment while allowing greater through-put by means of multitasking. In the first part of this book you will see how this near-magical combination is achieved.

OS/2 opens the doors to a whole new world for programmers. Fully harnessing OS/2's capabilities will allow you to create highly efficient and powerful programs, the likes of which could never be seen in either a DOS or UNIX-like environment. Frankly, from my point of view, OS/2 is the platform on which the next generation of applications will be built.

As you will see in this book, there is little in OS/2 that is difficult to grasp or use. However, OS/2 is so large that it is sometimes hard to see the larger view. (For example, version 1.1 of OS/2 contains several hundred different system services!) As you begin to learn to program for OS/2, it may seem hard to pull all the pieces together, but as you become more experienced, the logical design of OS/2 will become apparent. This book can help you achieve that "view from a height."

Part One of this book gives you an overview of OS/2's design philosophy. Part Two covers the most important core system services provided by OS/2. As you may know, OS/2 actually consists of two "pieces": the core (sometimes called the kernel) and the Presentation Manager. Most of this book deals with the core of OS/2 because it forms the logical starting point. A firm knowledge of the core services is necessary before progressing to the Presentation Manager, which is introduced in Part Three.

I used Microsoft C 5.1 to compile and test all the C examples in this book. (I compiled the two assembly code examples using Microsoft's MASM 5.1.) Although Microsoft C 5.1 is certainly a fine compiler, I used it out of necessity: it was the only compiler available for OS/2 when this book was being written. However, the code in this book conforms to the proposed ANSI standard and should be able to be compiled by virtually any OS/2-compatible compiler. (Keep in mind that certain Microsoft supplied **typedef**s may be given different names by other manufacturers.)

This book assumes that you have some experience as a programmer and a basic understanding of the PC hardware environment. Most of the examples are in C. If you are not as proficient in C as you would like, Appendix C presents an overview of the C language, which should be sufficient to help you understand this book. With few exceptions, C is and will be the high-level language of choice for OS/2 development.

This book includes many useful and interesting example programs. If you're like me, you probably would like to use them, but hate typing them into the computer. When I key in routines from a book, it always seems that I type something wrong and spend hours trying to get the program to work. For this reason, I am offering the source code on diskette for all the functions and programs contained in this book for $24.95. Just fill in the order blank on the next page and mail it, along with your payment, to the address shown. Or, if you're in a hurry, just call (217) 586-4021 to place your order by telephone. (VISA and MasterCard accepted.)

HS
Mahomet, Illinois
June 1988

ORDER FORM

Please send me _____ copy(ies), at $24.95 each, of the source code for the programs in *OS/2 Programming: An Introduction.* (Foreign orders, please add $5 shipping and handling.)

Name _____

Address _____

City _____ **State** _____ **ZIP** _____

Telephone (_____) _____

Diskette size (check one): 5 1/4" _____ 3 1/2" _____

Amount of payment: $ _____

Method of payment: check _____ VISA _____ MC _____

Credit card number: _____

Expiration date: _____

Signature: _____

Send to: Herbert Schildt
RR 1, Box 130
Mahomet, IL 61853
or phone: (217) 586-4021

I

INTRODUCTION TO OS/2 PROGRAMMING

Part One presents some necessary background information on OS/2 and discusses the special 80286 features that OS/2 takes advantage of. You will learn about OS/2's design philosophy and be introduced to OS/2's call-based interface.

1

OS/2: AN OVERVIEW

OS/2 is a very large program that consists of many subsystems. Although no single part of OS/2 is difficult to understand or use, it can be difficult to grasp the totality of the operating system. To help ease the problem, this chapter presents an overview of OS/2, including its design philosophy, operation, and basis in the 80286 processor. Many of the topics discussed in this chapter will be fully explored in subsequent chapters.

The chapter begins with a brief description of the origins of OS/2, followed by a discussion of the 80286 CPU, whose operation is so important to an understanding of OS/2 programming. The chapter concludes with a brief tour of the OS/2 programming environment. Several new terms that have been coined or popularized as a result of OS/2 are introduced along the way. If you already have a good basic understanding of the 80286 and OS/2's operation, you can skip to Chapter 2.

THE HERITAGE OF OS/2

Although OS/2 was created new from the ground up, it owes much to the operating systems that preceded it. To understand why certain things in OS/2 are the way they are requires that you understand OS/2's heritage. Those of you who participated in the microcomputer revolution of the late seventies already know much of the story. If you are new to microcomputers, however, many of the bits and pieces of OS/2 make the most sense when you understand where they came from.

Real microcomputer operating systems began with Digital Research's CP/M, which was designed for the Intel 8080 CPU, an 8-bit processor. (The 8080 was the forerunner to the 8086.) In the early days of microcomputing, each computer manufacturer supplied its own operating system, which usually consisted of little more than a primitive set of disk file I/O functions. In addition to being very crude, these operating systems suffered from the fact that they were different from each other. The differences between the systems prevented software developers from developing programs that could be mass marketed to the full range of microcomputers. When Gary Kildall, the founder of Digital Research, created CP/M it was with the goal of providing a common operating system for all microcomputers. To a great extent he succeeded in this goal. CP/M is a compact yet highly adaptive single-tasking operating system that was nothing short of perfect for the first 8-bit microcomputers.

The CP/M system is so important because it made all the various microcomputers' software compatible. Compatibility was a crucial, necessary ingredient for the future success of the microcomputer because it allowed software developers to invest large amounts of time and money in creating products that ran under CP/M. Without the unifying force of CP/M the software market would have been fragmented, and the cost-effective development of excellent software would have been impossible. As you will soon see, the issue of compatibility plays an important role in the development of OS/2.

When IBM began developing its first personal computer, the designers chose to base its architecture on the next generation of Intel microprocessors. These processors included the 16-bit 8086 and its close relative, the 8088. (IBM actually used the 8088 because it provided a cost-effective way to access a 16-bit processor using 8-bit interface chips. Hereafter in this book, a reference to the 8086 implies both the 8086 and the 8088.) Before the PC was released, experts speculated that it would use a new version of CP/M as its operating system. However, for reasons that are still unclear, Digital Research and IBM did not come to an agreement to use CP/M. Instead, IBM asked Microsoft, which was already working on languages for the PC, to develop a new operating system. The operating system was called PC DOS when first released. Now it is generally called DOS.

Because IBM and Microsoft knew that literally thousands of programs originally written for CP/M would be converted to run under DOS, DOS was designed to be highly compatible with the original

CP/M. In fact the basis for the file system and its system interface was CP/M. Like CP/M, DOS is a single-tasking, highly adaptive operating system that could fully control the new 16-bit microcomputers. Since its release in 1981, DOS has become the world's most popular operating system, with well over ten million users worldwide. Some analysts suggest that DOS will still be in common use into the twenty-first century.

As good as DOS is, it does suffer from two major shortcomings.

1. Because it was originally designed for use with the 8086, DOS can directly access only 1 megabyte of RAM. Within this megabyte, only 640K can effectively be used because of the way the ROM and video RAM of the original PC were located. Although 640K of program memory space still sounds like a lot when viewed from the perspective of many existing DOS applications, it is far too little for the next generation of "smart" (AI-based) software or for large database or spreadsheet programs. And 640K is not a lot of memory when it is used in a multitasking environment.
2. DOS is single-tasking. Without multitasking capabilities it is impossible to make the most efficient use of the computer. As you will see later in this chapter, much of the CPU's time is spent waiting for things to happen. During these "dead" moments, a multitasking operating system can run another task. In a single-tasking system, this time is simply lost.

The memory restriction found in DOS is based on the architecture of the 8086 processor and is not easily removed. Although it is possible to multitask the 8086, it is not a good idea because the 8086 provides no way to protect one task from another. That is, if two programs are executed simultaneously using an 8086 processor, one program could adversely affect the execution of the second. Thus DOS continues to limit application memory to 640K and to remain single-tasking. It was clear that any efforts to remove these restrictions would come about because of an advance in CPU design.

The next processor released by Intel was the 80186, which was really just a faster 8086 and not important otherwise. However, in 1984 Intel released the 80286. The 80286 CPU could run all programs written for the 8086 but included several new instructions and a second

mode of operation. When the 80286 was running in this second mode, it can address 16 megabytes of RAM and isolate concurrently executing programs from each other. The 80286 included two modes of operation for the sake of compatibility with software written for the 8086. However, both modes of operation are more or less mutually incompatible. These two modes are the cause of numerous problems as they relate to the creation of OS/2.

The 80286 is the processor that IBM chose to use in the PC AT. However, because no software existed to make use of the 80286's second mode of operation, it was run by DOS as simply a faster 8086, with all of its limitations.

The newest Intel processor in common use is the 80386, which is an improved version of the 80286. The 80386 includes three basic modes of operation: 8086 emulation, 80286 emulation, and its own 80386 operation. Actually, because of the way the 80386 is designed it does not truly have an 80286 emulation mode. More accurately, the 80386 automatically acts like an 80286 when presented with 80286 instructions. At the time of this writing, OS/2 runs the 80386 as if it were an 80286. However, an 80386 version of OS/2 is expected soon.

In 1987 IBM released its PS/2 line of personal computers. Although the low end of these systems is based on the older 8086 processor, the models 50 and 60 use Intel 80286, and the model 80 uses the 80386 processor. To take full advantage of these machines, a new operating system was required. Three of the most important goals in designing the new system were to eliminate the 640K memory barrier, to support multitasking, and, for better or worse, to provide an upward compatibility path from DOS. Toward these ends Microsoft and IBM launched a joint development project headed by Gordon Letwin on the Microsoft side and Ed Iacobucci on the IBM side. The result of their efforts is, of course, OS/2.

Simply stated, OS/2 is very likely the largest, most complex piece of software ever written for a microcomputer. It is also one of the most fascinating. To get an idea of its complexity, consider this: It took Microsoft about four months to develop DOS version 1.0; OS/2 has already taken three and a half years! The specifications for DOS 1.0 were about 100 pages long; it takes over 1500 pages to describe OS/2!

From this historical perspective, let's see what OS/2 is all about.

THE 80286 AND OS/2: A FAMILY AFFAIR

First and foremost, OS/2 is an 80286-based operating system. (Although OS/2 can also use the 80386, it does so as if the 80386 were an 80286.) In many ways OS/2 is the actualization of the imaginary operating system for which the designers created the 80286. The Intel designers created the 80286 for a multitasking environment. The fact that it could emulate its forerunner, the 8086, was a necessary but uninteresting dead end. What the designers created was a processor that could provide a solid base for the next generation of microcomputer operating systems. Toward this goal they implemented several important features that essentially defined what that operating system would be like. In fact, many of OS/2's features are closely linked with related features of the 80286. Hence the programmer's understanding of OS/2 really begins with an understanding of the 80286 processor.

Because of its heritage and attempts to maintain software compatibility with its ancestors, the 80286 is a somewhat "quirky" chip. This section will discuss some aspects of the 80286 that relate specifically to OS/2 programming.

Note: Nothing in the sections that follow assumes that you have significant familiarity with 80286 assembly language programming. However, implicit in OS/2 programming are the concepts of subroutines (both calling and returning from them), the stack, and stack operations. You should have at least a general understanding of how a computer goes about its business.

The Architecture of the 80286

The 80286 contains 14 registers into which information is placed for processing or program control. The registers fall into the following categories:

- General-purpose registers
- Base pointer and index registers

- Segment registers
- Special-purpose registers

All the registers in the 80286 CPU are 16 bits (2 bytes) wide.

The *general-purpose registers* are the "workhorse" registers of the CPU. It is into these registers that values are placed for processing, including arithmetic operations such as adding or multiplying; comparisons such as equality, less than, and greater than; and branch (jump) instructions. Each of the general-purpose registers can be accessed either as a 16-bit register or as two 8-bit registers.

The *base pointer* and *index registers* provide support for such things as relative addressing, the stack pointer, and block move instructions.

The *segment registers* support the 80286's segmented memory scheme. (The segmented architecture of the 80286 is discussed later in this chapter.) The **CS** register holds the current code segment, the **DS** holds the current data segment, the **ES** holds the extra segment, and the **SS** holds the stack segment.

Finally, the *special-purpose registers* include the flag register, which holds the status of the CPU, and the instruction pointer, which points to the next instruction for the CPU to execute.

Figure 1-1 shows the layout of the 80286 registers.

The Segmented Memory Model

The entire Intel CPU line is based on the original 8086, which views the memory of the system as if it were organized into 64K chunks called *segments*. Although we will examine more fully how the 80286 calculates the actual linear address of a specific byte in RAM, loosely speaking, what happens is that the contents of a segment register are combined with the contents of another register (or immediate value). This second value is called the *offset*, and the entire scheme is often called the *segment:offset* form of memory addressing.

Like most things, the segment:offset memory model has its good and bad points. In the plus column, the segmented scheme makes it easy to write relocatable code and makes it easier to develop virtual memory techniques. (OS/2 puts these features to good use, and you will read more about them later.) If used correctly, segmentation can also make some types of programs execute very quickly because the

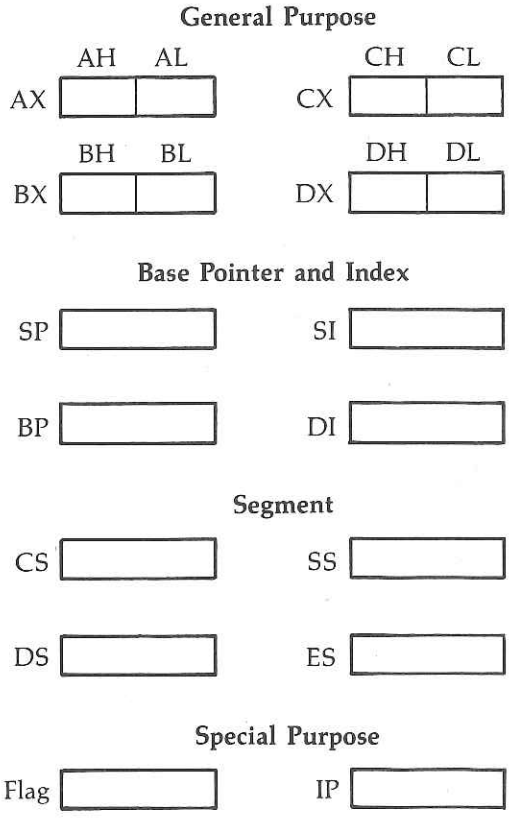


Figure 1-1. The 80286 CPU registers

segment registers can be loaded once and thereafter only the offset values need to be used to access memory, saving the time it takes to load a segment register repeatedly. In the minus column, the segmented approach tends to complicate what is essentially a nearly intuitive concept: memory. Most programmers, even long-time 80286 programmers, think of memory as strictly linear. This is the most natural view. However, the segmentation model requires that you think of memory abstractly, as disjointed pieces, a somewhat unnatural process.

On a more practical side, the segment:offset approach makes it significantly more difficult to create and access objects that require more space than is available in a single segment. The debate over the segmentation memory model has raged for years and will probably continue to do so. However, since this is the processor you have to work with, there is little use in worrying about its approach to memory. (In fact, because of the work that OS/2 does for you, you will not need to worry too much about where your programs execute in memory or how that memory is organized.)

One further complication concerns the segmented architecture of the 80286: The interpretation of the values contained in the segment registers varies between two 80286 modes of operation. These modes and the differences between them are the subjects of the next section.

The Two 80286 Modes of Operation

As you may already know, to maintain software compatibility with its ancestors, the 80286 has to be able to execute 8086 programs. To provide for this the 80286 CPU can operate in two distinctly different ways. In fact, the modes of operation are so different in some respects that it may be easier to think of the 80286 as two CPUs in one package. It is important to understand the differences between these modes of operation to grasp some of the subtleties of OS/2.

The two 80286 modes of operation are called *real* and *protected.* When the 80286 begins execution, it uses real mode by default. Real mode is essentially the 80286's 8086 emulation mode. When the 80286 is running in real mode, its address space is the same as the 8086's and is limited to 1 megabyte. Since the 8086 was not designed with multitasking in mind, any piece of code can issue any instruction and access any part of memory. Put in somewhat simple terms, in real mode what your program sees is what it gets. The name *real mode* is derived from the fact that a program is actually using real memory addresses when it accesses memory. That is, the values of the segment and offset registers actually contain the physical address that will be the target of a load, store, call, or jump operation. This is the mode for which DOS was designed.

When the 80286 is running in *protected mode* several new instructions become available, and the way the system memory and resources are accessed changes. Perhaps the most significant difference is how physical memory addresses are calculated. Because of the way addresses are

calculated in protected mode, the 80286 can directly access up to 16 megabytes of system RAM and up to 1 gigabyte of virtual address space. In protected mode, programs are assigned a privilege level. Only the most privileged programs have access to certain instructions, such as interrupt and I/O instructions. In protected mode it is also possible for the CPU to prevent one program from accidentally interfering with another that is concurrently executing. (This feature gave protected mode its name.) Finally, protected-mode operation allows the 80286 to use some special instructions that make multitasking easier to implement.

As far as the programmer is concerned the most important difference between real and protected modes is the way memory addresses are calculated. The next two sections explain both ways.

Address Calculation in Real Mode To access a megabyte of RAM requires at least a 20-bit address. However, in the 80286 no register is larger than 16 bits. This means that the 20-bit address must be divided between two registers. Unfortunately, the way the 20 bits are divided is a little more complex than one might assume.

In real mode, all addresses consist of a segment and an offset. A segment is a 64K region of RAM that must start on an even multiple of 16. In 80286 jargon, 16 bytes is called a *paragraph;* you will sometimes see the term *paragraph boundary* used to reference these even multiples of 16 bytes. The 8086 has four segments: one for code, one for data, one for stack, and one extra. The location of any byte within a segment is called the offset. The actual 20-bit address of any specific byte within the computer is the combination of the segment and the offset.

To calculate the actual byte referred to by the combination of the segment and offset, first shift the value in the segment register to the left four bit positions and add this value to the offset. This makes a 20-bit address. For example, if the segment register holds the value FFH and the offset holds AH, the following sequence shows how the actual address is derived. The absolute 20-bit address is 300H.

segment register:		0000	0000	1111	1111
segment shifted:	0000	0000	1111	1111	
offset:		0000	0000	0000	1010
segment+offset:	0000	0000	1111	1111	1010

The resulting 20-bit address is FFAH. However, you will almost never see a real-mode address referred to in this form. Instead, the segment:offset form is used. In this case the address would be written 00FF:000A. Many segment:offset addresses can describe the same byte because the segments may overlap each other. For example, 0000:0010 is the same as 0001:0000.

Address Calculation in Protected Mode When the 80286 is running in protected mode, memory addresses are computed in a fundamentally different way from that used by real mode. Although memory is still accessed via the segment:offset combination, the meaning of the 80286 segment registers has been altered. In protected mode the segment registers hold an index into a table, which holds the physical address of an object in memory. In 80286 jargon, this table is called a *segment descriptor table* or *descriptor table* for short. To repeat, in protected mode the value of a segment register no longer refers to a physical memory address. Rather, its value is used as an index into a descriptor table. For this reason, when the 80286 is running in protected mode, the segment registers are sometimes called *selectors*.

Each entry in a descriptor table contains at least three items of information. The first is a 24-bit value that is the base address of the segment in question. This value points to the start of a segment in much the same way that the value of a segment register does when the CPU is executing in real mode. However, since 24 bits are provided, it is possible to access up to 16 megabytes of memory directly, which surpasses the single megabyte limit found in real mode.

The second item of interest stored in the table entry is the size of the segment. This is a 16-bit value, which means that segments can be up to 64K in length. In protected mode the size of a segment may vary; in real mode it is fixed at 64K. The length information is used to prevent one program from accessing memory that it shouldn't. If an attempt is made to access memory outside a program's allocated memory, the 80286 generates a general protection fault that returns control to OS/2.

Finally, the table includes an 8-bit access rights entry. These rights include read/write access, execute only access, present or absent indication, and a privilege level. It is possible to mark a memory segment for read only access if it is a data segment or for execute only access if it is a code segment. The 80286 allows programs to be given different privi-

lege levels, going from most trusted (level 0) to least trusted (level 3). OS/2 uses all the information stored in the access rights field to support multitasking and virtual memory.

When the 80286 calculates an address in protected mode, it uses the value of a segment register as an index into a descriptor table. It then adds the base segment address to the offset to provide the final physical address. This process is depicted in Figure 1-2. As the address is being calculated, the access information is being checked. If your program attempts to reference memory that it shouldn't, a general protection fault will be generated.

Remember that the resolution of a memory address is done for you by the CPU and requires nothing on your part.

The 80286 maintains three types of descriptor tables: the *global descriptor table (GDT)*, the *local descriptor table (LDT)*, and the *interrupt descriptor table (IDT)*. In general, the GDT holds address information that is available to all tasks in the system; the LDT holds address information that

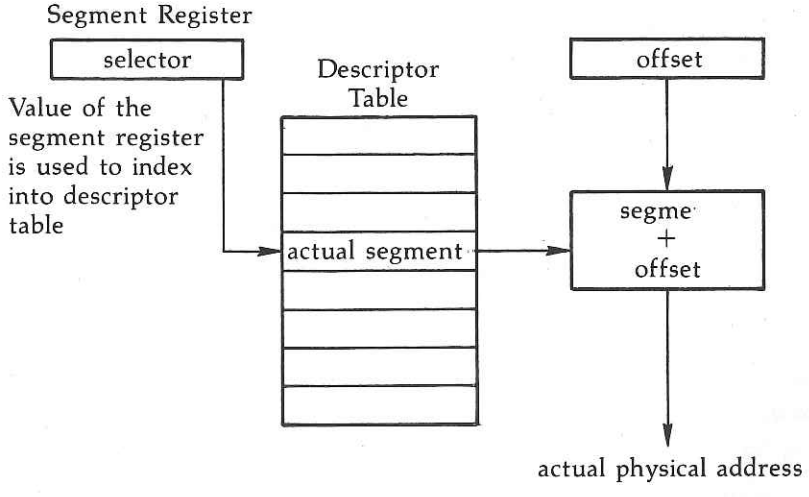


Figure 1-2. Address calculation in protected mode

is local to each task; and the IDT holds address information related to the interrupt service routines. As stated, OS/2 maintains these tables automatically. As a rule you don't need to worry about them while programming, but knowing their functions is important for a clear understanding of how OS/2 handles multiple tasks.

The Advantages of Protected-Mode Addressing

Aside from the fact that a larger amount of memory can be accessed in protected-mode operation, the use of descriptor tables and the change in the meaning of the segment registers have several positive effects that OS/2 capitalizes on to provide a stable and efficient multitasking environment.

- Because the segment register holds an index rather than an address, the operating system can move segments about in memory at will by changing the base segment address in the descriptor table entry. This is accomplished completely invisibly to the application program because the program does not "know" what part of memory it is using. Thus, even while the program is executing, it can be moved about in memory. This feature is important because it allows tasks to be swapped in and out of memory. Thus it is possible for OS/2 to *overcommit* memory by moving tasks in and out of RAM and storing them temporarily on disk. This means that you can run programs that require more RAM than the system has or to run more programs simultaneously than would normally fit in the system RAM.
- The fact that the size of a segment is stored in the descriptor table prevents programs from interfering with each other. Although it is certainly possible to multitask in real mode, it is very dangerous to do so because programs can access any location in memory. To be stable, a multitasking environment must have a means of preventing one program from destroying another program's code or data. The segment size entry helps accomplish this.
- The fact that various access rights, including privilege levels, are now linked with a memory location allows OS/2 to control access both to itself and to other system resources. Essentially, for code to access memory it must have equal or higher access privileges. The effect of privilege levels will be discussed further a little later in this chapter.

Call Gates

In the foregoing discussion of memory access under protected-mode operation it may have occurred to you that a CALL is also affected by the change in the way the segment register is used. There are two basic types of CALL instructions: NEAR and FAR. A NEAR CALL is one that calls code in the same segment as the caller. A FAR CALL calls code that lies in a different segment from the caller. Whenever FAR calls are made, there must be some way to determine the actual address of the routine. This is accomplished by using a *call gate,* which is a special type of entry in either the global or local descriptor tables.

A call gate entry contains, among other things, the segment selector and offset of the called routine. This means that no offset information need be known by the calling routine. The only information the calling routine needs to know to execute a FAR CALL is the index of the call gate in the descriptor table. As you can probably imagine, this makes it easy to relocate code inside the memory of the computer, even while a program is executing. OS/2 simply needs to move a routine and update its address in the table. Since the index in the table remains the same, your program never knows that the target routine has been moved. As is the case with other memory accesses, the calculation of the actual physical address is performed by the CPU and is invisible to the programmer.

As was explained in the previous section, accesses to memory are controlled by privilege level. In a similar fashion, calls to a subroutine are executed only if the called routine is at the same or a lower privilege level than the caller. This feature is included in the 80286's protected-mode operation to prevent one program from interfering with another. However, a problem arises when a less-privileged routine needs to call a more-privileged one for legitimate purposes. The 80286 implements a solution to this problem by using call gates. A call gate can be used to allow a less-privileged routine access to more-privileged ones. As you will see, this is a very important feature as far as OS/2 is concerned.

One field in the call gate's descriptor table entry is its privilege level. A call gate may be called only by a program that is at least as privileged as the gate. However, the gate can "pass along" a call to a more-privileged routine. (Exactly how this is accomplished by the 80286 is a bit complicated and not important to programming for OS/2. The interested reader should refer to the various 80286-related publications

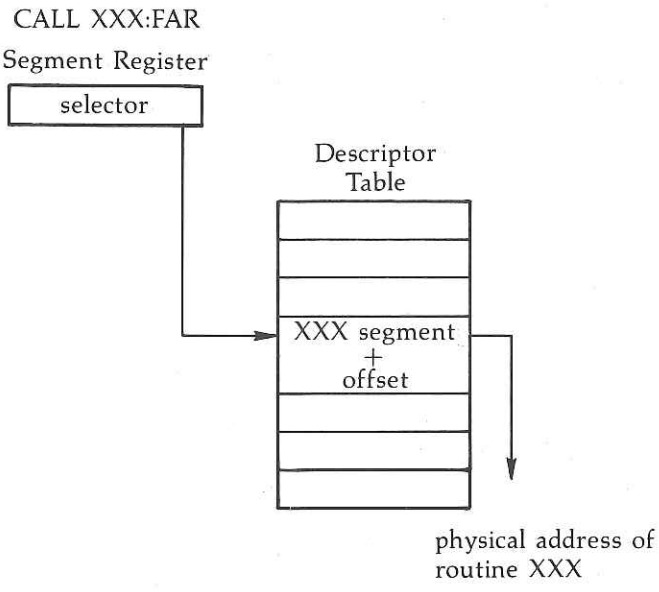


Figure 1-3. Using a call gate

by the Intel Corporation.) Essentially, the use of a call gate allows a more-privileged piece of code to be accessed in a carefully restricted way by a less-privileged program.

The function of a call gate is illustrated in Figure 1-3.

I/O Privileges

Another feature of the 80286 is its I/O protection. Because the protected-mode operation of the 80286 was designed for a multitasking environment, it had to have some way of controlling access to certain instructions, including input and output instructions. (Without this control, several different applications could—and probably would—write to the same devices at the same time, resulting in chaos.) Control is achieved via a program's *I/O privilege level* (IOPL for short). Although

the details are not important for the purposes of this book, the basic IOPL concept works as follows. The only routines that have access to IN and OUT instructions—and to the various interrupt instructions—are the routines that have been granted I/O access. (OS/2 has a facility that allows your programs to perform I/O operations directly, instead of using an OS/2 system call, in the few cases where it is really necessary.)

OS/2 and the Two 80286 Operational Modes

The 80286 mode of operation designed for a multitasking environment is the protected mode. Hence OS/2 uses this mode and requires all programs that execute under its control to do likewise. There is, however, one annoying exception: the DOS emulator.

Although OS/2 is a protected-mode operating system, the OS/2 designers needed to provide what is sometimes called a *compatibility path* from the older DOS to OS/2. Toward this end they needed to create a DOS emulator to run more or less under the control of OS/2. However, DOS is a real-mode operating system. Real mode and protected mode are mutually exclusive; they can't both be active at the same time. Here is the solution the OS/2 developers chose: When running a DOS program, use real mode; when running an OS/2 program, use protected mode. Although this solution sounds benign on the surface, it was devilishly difficult to implement, as you will see.

The first problem: Not only are real and protected modes incompatible, but also no instruction exists to switch from protected to real mode! When the 80286 is first turned on, it is in real mode. This approach is used to maintain compatibility with the 8086. There *is* a way to switch the 80286 into protected mode, but when it was designed no one thought that, once in protected mode, there would ever be a reason to switch back to real mode. As it turns out, the only way to switch from protected mode to real mode is by executing what amounts to a full system reset!

The second problem: A real-mode program can take full control of the system, bypassing any operating system that is present in the system. As you will soon see, OS/2 must control all system devices if it is to keep multiple tasks from trying to use the same device at the same time. This control is achieved largely through the use of the protected mode's privilege and I/O protection levels, which do not exist in real

mode. Although OS/2 can prevent some types of device request collisions, it cannot stop them all. As part of its solution to this problem, OS/2 fundamentally treats real-mode programs differently from protected-mode ones. In fact, the DOS emulator and the programs that execute under the emulator are given the lower 640K of RAM in the system. OS/2 and its applications use RAM from 1 megabyte up. In this way, no real-mode application can access any protected-mode application's code or data because the largest address reachable by a real-mode application is 1 megabyte. (It is possible to configure OS/2 so that no real-mode applications are allowed. In this case the first megabyte of RAM is also usable by OS/2.)

OS/2 ESSENTIALS

From a programming perspective, the most important attribute of OS/2 is its multitasking capabilities. Virtually all differences between DOS and OS/2, for example, are due either directly or indirectly to OS/2's support of multitasking.

Threads, Processes, and Tasks

The OS/2 design team did multitasking right! OS/2's tasking model is based on the simultaneous execution of pieces of code rather than on the simultaneous execution of programs. In OS/2 terminology, the smallest unit of execution is called a *thread.* All programs consist of at least one thread and may contain several. Hence, it is possible for a single program to have two or more parts of itself executing at the same time. This means that not only can OS/2 execute two or more programs at the same time, but that it can also execute two or more parts of a single program concurrently.

In OS/2 terminology a *process* and a *task* are the same and they are very loosely synonymous to the term *program.* A process owns various resources, including such things as memory, files, and threads.

The OS/2 Multitasking Model

As OS/2 is currently implemented, it is designed to share a single 80286 among several threads. It does this by granting each thread a short amount of CPU time, called a *time slice.* Although technically speaking only one thread actually executes at a time, the time slicing is

so rapid that the threads in the system appear to be running at the same time.

OS/2 controls multitasking by using a preemptive, priority-based scheduler. OS/2 associates a priority with each thread. Higher priority threads are granted access to the CPU before lower priority ones. There are three main priority categories. In order of highest to lowest, they are

- Time-critical
- Regular
- Idle

Time-critical tasks are tasks that must respond immediately to some event, such as communication programs. There are 32 priority levels within the time-critical category.

There are really two kinds of regular tasks. When a program is on the screen, OS/2 gives its threads a foreground priority, which is the highest priority a regular task can have. This is done to ensure that interactive sessions always take place without jerky or sluggish responses. Other regular threads in the system are given background priority when they are not displayed on the screen. Within this level there are 32 priority levels. OS/2 dynamically changes the priority of nonforeground threads at this level to use the CPU most efficiently.

The lowest priority tasks are given idle priority. This level executes only when there are no higher priority tasks capable of executing. There are 32 priority levels within this group.

OS/2 always runs the highest priority thread capable of executing. When two or more threads share the same priority level, they are granted CPU time slices in a round-robin fashion. You may think that a high-priority thread will dominate the CPU, but this is not the case because most programs, even time-critical ones, spend much of their time waiting for an event to occur. When a thread is waiting, OS/2 stops executing it and runs another. OS/2 also has certain parameters that determine the longest amount of time a process can be suspended.

A thread inside a process is in one of three mutually exclusive states: blocked, ready-to-run, or running. Any time a thread is waiting for something, its execution is said to be blocked. For example, a thread that is part of an interactive program may be waiting for keyboard input. Until that input is achieved, the thread can execute no further,

and the execution of that thread is blocked. Blocked threads are not given CPU time until the event they are waiting for occurs. Once this happens, the thread is in a ready-to-run state, but it is still not executing. It resumes execution only when OS/2's scheduler grants it a slice of CPU time. If the unblocked thread is of higher priority than the thread currently being executed, the currently executing thread is preempted and the unblocked thread is allowed to run. Otherwise, it must wait until all higher priority tasks are blocked.

The single most advantageous attribute of a thread-based multitasking system is that it allows greater throughput because independent pieces of your program can execute concurrently. For example, a word processing program could simultaneously format text for output and take input from the user. Later in this book, substantial space will be given to multithread programs.

Interprocess Communication

OS/2 supports several forms of *interprocess communication* (IPC). These include pipes, queues, semaphores, signals, and shared memory. Many devices are sequential in nature; that is, they cannot be used by two or more threads at the same time. Whenever two or more threads need to use one of these devices, they must coordinate their activity. The part of a program that accesses such a device is called a *critical section*. Before entering a critical section a thread must make sure that the device accessed by that section is not already being used by another thread. This is accomplished by using IPC, and the process is called *synchronization*. You will see several examples of this.

OS/2's Protection Strategy

As mentioned earlier during the discussion of the 80286, a successful multitasking operating system must prevent programs running under it from adversely affecting each other or the operating system itself. In essence, the operating system must protect programs and itself from harm. OS/2 achieves this protection by using the 80286's privilege level mechanism and protected-mode addressing scheme.

The 80286 supports four privilege levels: level 0 is the most trusted and level 3 is the least trusted. In OS/2, the core routines, usually called

the *kernel*, are at level 0. Level 1 is unused at this time. Level 2 contains the system services, and application programs run at level 3. The only way to access routines at a more trusted level is through a call gate. This is the method used by OS/2 to give your programs access to the various OS/2 services. OS/2 uses this scheme to prevent a program from accessing any part of OS/2 in an uncontrolled manner.

If a program attempts to access memory outside its currently defined segments, a general protection fault is generated. OS/2 intercepts this fault and terminates the process that caused it. In this way one program cannot destroy another's code or data areas. (Keep in mind that it is possible for two or more programs to share memory when that is desirable.)

Because OS/2 controls the descriptor tables, it can mark certain segments as read only, which means that programs can read the data in that segment but not change it. OS/2 can also mark a segment as execute only, which allows system routines to be used but not modified.

Finally, OS/2 has control of all I/O devices. This means that, in general, an application program cannot execute an IN or OUT instruction or turn interrupts on or off. (In a multitasking operating system all I/O is interrupt driven; hence a program cannot be allowed to alter the state of the interrupts.) By denying the use of I/O instructions, OS/2 prevents two or more programs from accessing the same device at the same time. (OS/2 can grant a program the ability to perform I/O in some special situations.)

Virtual Memory

OS/2 takes advantage of the 80286's virtual memory capabilities. OS/2 can overcommit the memory of the system by swapping unused segments to disk until they are needed. Although excessive swapping can bring a multitasking system to a crawl, a small amount of swapping is hardly noticeable because most programs contain code that is seldom executed. When a request for memory is made and none is available, OS/2 examines each segment and swaps to disk the one least recently used. Should this memory be needed, a memory fault is generated and OS/2 swaps the segment back in, perhaps removing a different segment in the process. What is particularly nice about OS/2's virtual memory capabilities is that they are performed automatically and do not require any additional effort on your part.

THE APPLICATION PROGRAM INTERFACE

A program accesses OS/2's system services via the Application Program Interface (API). Unlike its forerunner, DOS, OS/2 does not use a software interrupt scheme to use a system service. Instead the API is a *call-based interface.* In this approach, each OS/2 service is associated with the name that is used to call it. To use this method any necessary parameters are pushed onto the stack and the appropriate OS/2 function is called. For example, the OS/2 function **DosSleep** is used to suspend the execution of the thread that calls it for a specified number of milliseconds. Shown in pseudoassembly, this is how **DosSleep** is called so that the calling thread suspends for 100 milliseconds:

```
PUSH 100
CALL DosSleep
```

Most OS/2 functions return 0 in the **AX** register if successful.

If you are programming in a high-level language like C, the compiler puts the parameters to a call on the stack for you. However, if you are programming in assembler, your programs must do this explicitly.

It is possible to create programs that will execute in both DOS and OS/2 environments. However, these programs must use only those system calls that are part of the Family Application Program Interface (FAPI). This is a very restricted set of functions that are common to both DOS and OS/2.

DYNAMIC LINKING

The API is implemented in OS/2 by using a procedure called *dynamic linking.* Here is how it works. All the functions in the API are stored in a relocatable format called a *dynamic link library* (DLL). When your program calls an API function, the linker does not add the code for that function to the executable version of your program. Instead, it adds loading instructions for that function, such as what DLL it resides in. When your program is executed, the necessary API routines are also loaded by the OS/2 loader. (It is also possible to load routines after the program has started execution.) A dynamic link routine is called a *dynlink.*

Dynlinks have some very important benefits. First, since virtually all programs designed for use with OS/2 will use OS/2 functions, the use of dynlinks prevents disk space from being wasted by the significant amount of duplicated object code that would be created if the OS/2 function code were actually added to each program's executable file. Second, updates and enhancements to OS/2 can be accomplished by changing the dynlink libraries. Thus existing programs automatically make use of the improved or expanded functions. Finally, it is possible for you to create your own dynlink libraries and let your programs receive the preceding advantages.

THE PRESENTATION MANAGER

Although not included in OS/2 version 1.0, the Presentation Manager is a standard part of OS/2 beginning with version 1.1, and all users with 1.0 received upgrades that included the Presentation Manager. The Presentation Manager is a top-level graphical interface that resembles Microsoft Windows version 2.0. It supports such things as multiple overlapping windows, various character fonts, menu selections, and the mouse. The Presentation Manager will be introduced later in this book, after you have mastered the basics of OS/2 programming.

THE DOS-OS/2 TUG-OF-WAR

As you have probably gathered from reading this chapter, DOS—and the DOS emulator—are at odds with OS/2. The resolution of their incompatibility was not 100 percent achieved. Consider this: High-performance DOS programs gain that performance by *bypassing* DOS. This clearly violates the basic philosophy behind OS/2, in which the operating system *must be in control.* Therefore, some DOS programs simply will not run under the OS/2 DOS emulator.

Because of the fact that DOS programs typically perform direct device I/O, OS/2 allows DOS programs to run only when they are on the screen (foreground mode). This means that when you are running a DOS program and an OS/2 application and you have the OS/2 application on the screen, the execution of the DOS program is suspended. OS/2 allows only one DOS emulator to be active in the system, and it can run only one DOS program at a time.

As was mentioned in the API discussion, a subset of the API, called the FAPI (Family Application Program Interface) can be used to create programs that execute under both DOS and OS/2. Although this is convenient for a small group of applications, it will probably not be very important in general because the FAPI supports such a restricted set of OS/2 functions. More likely, separate DOS and OS/2 versions of programs will continue to exist.

As you have seen, it is possible to insulate DOS applications from OS/2 applications to a great extent, but not 100 percent. For this reason it is possible for a DOS application to crash the computer it is running on. (OS/2 is supposed to be crash proof because of the protected memory scheme, although any bugs in OS/2 could, of course, cause a system crash.) Because of these types of basic incompatibilities, the use of the DOS emulator will decline rapidly once new OS/2-specific versions of programs begin appearing.

As the title implies, this book is about OS/2 programming. The main emphasis will be on the OS/2 protected-mode environment, the API, and the Presentation Manager.

THE OS/2 PHILOSOPHY

Embodied in the functional aspects of OS/2 is the OS/2 philosophy: OS/2 should provide a stable multitasking environment that is both flexible and extensible. As you have seen, the 80286 supplies the raw material to support a stable multitasking environment in which one program cannot destroy another. Its protected-mode addressing scheme allows OS/2 to support dynamic linking, which allows easy modification of most of OS/2's code. It also allows new OS/2 system services to be added by either Microsoft, IBM, or a third party.

From the programmer's point of view OS/2 is a giant toolkit. In the rest of this book you will learn how to access those tools to create OS/2 programs.

2

OS/2 INTERFACING FUNDAMENTALS

This chapter will examine in significant technical detail several key points relating to the use of OS/2's Application Program Interface (API) services. The API services are your program's gateway to OS/2. Before you can begin to write programs that run under OS/2, you need to understand exactly how to work with the API.

This chapter begins with a discussion of the OS/2 call-based interface. You will see how to compile (or assemble) and link OS/2-compatible programs. Along the way two sample programs illustrate several important OS/2 interfacing concepts. Finally, you will be introduced to the API dynlink library routines by category.

Although the rest of the examples in this book are in C, the examples in this chapter are shown in both C and assembly code. The reason for the assembly code examples is that they illustrate the process of interfacing to OS/2 on the actual machine instruction level. Even if you will never program for OS/2 using assembler, it is still valuable to understand exactly what the interfacing process is.

THE OS/2 CALL-BASED INTERFACE

Your program interacts with OS/2 by using the API dynlink functions. Chapter 1 mentioned that OS/2's API functions are accessed via a CALL instruction, and a very general explanation of the procedure was given. Here, you will learn in detail how to call the API routines.

The Call Format

Routines in the API (or any dynlink library, for that matter) must be reached by issuing a FAR call instruction. Remember that a FAR call instruction is used when the called routine is in a different segment from the calling routine. (The opposite of a FAR call is a NEAR call, which is used for intrasegment CALL instructions.) Before issuing the CALL instruction, however, your program must push onto the stack, in the proper order, the parameters used by the API service you will be calling. The OS/2 API interface supports four different types of parameters:

1. byte
2. word
3. double word
4. pointer (address)

Before discussing these further, let's take a short detour and review the difference between call-by-reference and call-by-value parameter-passing conventions.

Call-by-Value There are essentially two ways in which a subroutine can be passed its parameters. The first is *call-by-value.* Using this method the subroutine is passed copies of the actual information (values) it needs. Any modifications the subroutine makes to a parameter's value do not affect the calling routine's copy of the parameter; the subroutine is always operating on a copy of the original value.

Call-by-Reference Parameters can also be passed to a subroutine through *call-by-reference.* In this approach the calling routine passes to the subroutine the address of (in C terms, a *pointer* to) each parameter. When this method is used, the subroutine indirectly accesses and manipulates the original data found in the calling routine. Hence changes to the parameter affect the caller's copy because the subroutine is actually operating on the caller's data.

The OS/2 API services require the use of both call-by-value and call-by-reference. All byte values have only their addresses passed to the API. A word or double word can be passed either by value or by reference. If the API service does not need to return information to the

caller via a word or double word value, call-by-value is used; otherwise, the parameters are passed by reference. Any complex or variable length data structures must be passed by reference. Several of the API services operate on conglomerate data types that are the equivalent of a C structure. OS/2 does not pass these on the stack; it passes only a pointer.

Some API services use what is called an *ASCIIZ string*, which is simply a null-terminated (ASCII 0) string. When a string of this sort is required, only its address is passed, not the entire string.

Error Return

As stated in the preceding section, the OS/2 API functions return information to the calling routine through call-by-reference parameters. However, most of the API services return a success/error code in the **AX** register. When an API service is called from a C program, the value returned in the **AX** register automatically becomes the return value of the API routine. In general all the functions return zero when successful. A nonzero return implies an error.

AN ASSEMBLY CODE EXAMPLE

This short assembly language program illustrates how the two API services, **DosBeep** and **DosExit**, are called. The **DosBeep** service beeps the speaker at a given frequency for a given duration. Both parameters are word values; the frequency is pushed first, followed by the duration. **DosExit** is the standard OS/2 program termination function. Generally speaking, all OS/2 programs must end by calling **DosExit.** Its two-word parameters represent an action code and a result code. The action code is pushed first. If the action code is 0, only the current thread is terminated. If it is 1, the entire process is terminated. The value of the result code is returned to OS/2.

The program shown here uses **DosBeep** to produce a "whooping" sound by varying the frequency used to call **DosBeep** from low to high. The process repeats five times.

```
; A First OS/2 protected mode program.
;
; This program causes a "whooping" type sound using
; the speaker.

PAGE ,132                       ; set page dimensions
```

```
; Set up 16-bit segments
        DOSSEG                          ; Microsoft segment conventions
                                        ; for OS/2 protected mode
                                        ; programs

        .MODEL  SMALL                   ; set model size for program
        .286                            ; use 80286 instructions

        .STACK  300H                    ; set up 768 byte stack

        .DATA
DUR     DW      1
FREQ    DW      0
TIMES   DW      5

        .CODE
START:                                  ; beginning of code
        EXTRN   DOSBEEP:FAR, DOSEXIT:FAR
SPROC   PROC    FAR                     ; declare the main procedure

MORE:   MOV     FREQ,100                ; starting frequency
AGAIN:  ADD     FREQ,50                 ; frequency increment

        PUSH    FREQ                    ; DOSBEEP function parameters
        PUSH    DUR
        CALL    DOSBEEP                 ; call it

        CMP     FREQ,2500               ; upper frequency yet?
        JLE     AGAIN                   ; if not, do it again
        DEC     TIMES                   ; decrease count on repeats
        JGE     MORE                    ; if not zero, make sound again

        PUSH    0                       ; setup for exit
        PUSH    0
        CALL    DOSEXIT                 ; call API exit function
SPROC   ENDP                            ; end

        END     START
```

To assemble this file you will need an OS/2-based assembler and linker. One that will work is the Microsoft Macro Assembler version 5.1 (or later). Beginning with version 5.1, Microsoft has included OS/2 support and compatibility in its standard assembler package. If you use this package, the following commands will assemble and link the program. (Assume the program is called WHOOP.ASM.)

```
MASM WHOOP;
LINK WHOOP,,,DOSCALLS.LIB;
```

The file DOSCALLS.LIB is the library that contains references to the dynlink code for the API functions used in the program. (More about DOSCALLS.LIB in a moment.) No matter whose assembler and linker you are using, several assembler and linker options may be applicable to

some of the programs you write, so you must study your user manuals carefully.

Let's look closely at this program. First, the DOSSEG command is used to set up the program's segments in a manner consistent with OS/2's needs. The .MODEL directive tells the assembler the memory model you are using to compile your program. In this case the small model is used. The .286 directive lets the assembler know that 80286 instructions should be accepted. Notice that both **DosBeep** and **DosExit** are declared as FAR external procedures. Since both reside in a dynlink library and not in the program's source file, the assembler must be told to generate an external reference for them. Keep in mind that the EXTRN statement is used when a routine is found in a dynlink library, a regular library, or a separately compiled file. In this case, **DosBeep** and **DosExit** happen to be dynlink API services. Remember that all dynlink routines require a FAR call.

All OS/2 programs must define their own stack. This program creates one that is 300H bytes long. Although it was possible for sloppy programs to use the DOS system stack on many occasions, this is not the case with OS/2. Each thread must have its own stack to support multitasking. Keep in mind, however, that when you are using a high-level language, such as C, the compiler will automatically set up a stack for you.

In this simple program, the function **DosBeep** was assumed to be successful and its error return code is not examined. **DosExit** does not return a code to the program for obvious reasons. As you will see, many of the API services will either always work or always work if you supply correct input. For this reason the error code is often ignored in the interest of speed. As you will see in subsequent examples, however, certain API services should always have their return codes examined.

The API functions **DosBeep** and **DosExit** are found in dynamic link libraries. However, to add the correct dynlink loading information for those services, the linker needs to have the file DOSCALLS.LIB specified on the link line. DOSCALLS.LIB is a special type of library that contains information about how to load a dynlink routine rather than the actual code for the routine. This information includes the name of the routine plus the name of the file in which it is stored. (Remember that all dynamic link files end with the extension .DLL.) This information is put into your .EXE file, and the API services used by your pro-

gram are loaded when needed. Later in this book you will learn how to create your own dynlink libraries.

Because OS/2 is a new operating system, it is going through a period of frequent revisions and upgrades. Thus it is possible (but not likely) that certain filenames or function names could be changed in subsequent versions. For example, a later version of OS/2 might call DOSCALLS.LIB something else. Be sure to check your user manuals.

In assembly language programs, the names of the API services must appear in uppercase.

A C PROGRAM EXAMPLE

The C program that follows shows a slightly improved version of the WHOOP program. In this case, the program continues to make sounds until a key is pressed. To compile this program you must have a C compiler that runs under OS/2. Beginning with Microsoft version 5.1, the Microsoft C compiler can be run under OS/2. To compile the program use this command:

```
CL -Lp WHOOP.C
```

This causes the program to be compiled and linked, including the necessary dynlink libraries. The -Lp directive tells the compiler to produce a protected-mode program capable of being executed under OS/2. No matter whose C compiler you are using, several compiler and linker options may be applicable to some of the programs you write, so study your user manuals carefully.

```
/* C language demonstration program using DosBeep */

#include <os2.h>

main()
{
  register int i;

  for(;;) {
    for(i=100; i<2500; i+=50) {
      DosBeep(i, 1);  /* sound the speaker */
      if(kbhit()) break; /* look for keypress */
    }
    if(kbhit()) break; /* look for keypress here, too */
  }
  getch();  /* read and discard the keypress */
}
```

Because C is a high-level language, you call many of OS/2's functions only indirectly. For example, the **DosExit** function is called automatically when a C program terminates; you don't have to call it explicitly. Also notice that C's standard functions like **kbhit()** and **getch()** can be used. These functions in turn access the necessary API services. As you will see, there are some API services that you will not usually call directly, because they have direct parallels in the C standard library. However, there are circumstances in which you may want to call an API service even when a high-level language function can perform the same action because they often allow greater flexibility and control.

In the C environment, the API services are called using their mixed case version, such as **DosBeep**. In assembly language, however, the names must appear only in uppercase.

The header file OS2.H must be included with each C program or module. This file adds to your program all the information required to use the API services. (Your compiler may call this file something else, so check your user's manual.)

C AND THE API PARAMETERS

The following table shows the correspondence between the API data types and the C data types:

API	C
byte	char
word	unsigned
double word	unsigned long
address	(type far *)

Because all calls to the API are FAR calls, any address parameters used in an API call must also be FAR. In C this is accomplished in one of three ways.

1. You can explicitly define a pointer type as FAR by using the **far** C keyword. For example, this creates a FAR character pointer called ptr:

```
char far *ptr;
```

2. You can employ a type cast. This method is especially useful in connection with the **&** operator. For example, this expression generates a FAR address:

```
(char far *) &count
```

3. You can simply compile your program using one of the large code memory models. If you do this, all addresses are FAR by default.

This book will explicitly declare or cast all pointers to be FAR so that the code will run correctly under any memory model.

Pascal Versus C Calling Formats

A high-level language has two ways to push the arguments to a function on the stack. Pascal, for example, pushes the arguments on the stack in order from left to right. C normally pushes the arguments in order from right to left. All the API services must be called using the Pascal convention. For this reason, Microsoft C (and any other C compiler that supports OS/2) includes the function type modifier **pascal**. When **pascal** precedes a function's definition, the C compiler automatically uses the Pascal calling convention, thus matching with the API interface. All of the API services are declared as **pascal** in a C header file, and this file must be included with each program. In the C program just shown, the header OS2.H automatically includes all API declarations.

Since all calls to the API are FAR calls, each API service must also be declared to be FAR. Therefore, each API routine must be declared to be both **pascal** and **far** in the header file. For example, the **DosBeep** function can be declared like this:

```
unsigned pascal far DosBeep(unsigned, unsigned);
```

The API functions are often declared by using user-defined types. For example, the **DosBeep** function is declared by Microsoft like this:

```
USHORT APIENTRY DosBeep(USHORT, USHORT);
```

In the Microsoft header files **USHORT** is defined as **unsigned** and **APIENTRY** is defined as **pascal far**. Both forms mean the same; do not be confused by the type differences.

The fact that the API routines use the Pascal calling convention does not imply that Pascal is the best language to use for OS/2 programming. Indeed, it is quite the contrary! OS/2 is highly compatible with C. In fact, C is expected to be the dominant language for OS/2 development because it allows the greatest control and closest interaction with the API. C is also the most popular high-level language for PC software development. (Indeed, this is why it is used for the examples in this book.) However, for somewhat complex reasons, it was better to use the Pascal calling format for the API routines.

One final point has meaning mostly for assembly language programmers. In the Pascal calling convention, the called routine is responsible for removing the parameters from the stack. Since the API services use the Pascal convention, your routines need not remove the arguments that they pushed onto the stack.

A SHORT WORD ABOUT .DEF FILES

If you already know something about OS/2 programming, you may have heard about .DEF files. Essentially, a .DEF file is a text file that contains information about a source code file that you will be assembling or compiling. The .DEF files are used mainly to allow the creation of dynamic link (dynlink) libraries. Their use with nonlibrary code is optional, and no .DEF files are needed to assemble and run the sample programs just shown. Also, you do not need a .DEF file to *use* an existing dynlink library. You will learn more about .DEF files in the discussion of dynlink libraries.

CODE CONSTRAINTS

Code that is to be run under OS/2 is subject to a few constraints that did not apply to the old DOS environment.

- First and foremost, your code must be *reentrant*. A routine is said to be reentrant when it can be interrupted and executed by a thread

while it is being used by another thread. In essence, reentrant code is capable of being used by several threads at the same time.

- Your program cannot enable or disable interrupts, and it must not issue an INT instruction.
- Your programs must not attempt to alter the contents of a segment register or to perform segment "manipulations" as was commonly done when writing DOS programs. Basically, your program should let OS/2 manage memory.

ANOTHER SIMPLE EXAMPLE

For another example of interfacing to OS/2 via the API call-based interface, let's use the **VioWrtTTy** function to write a string of characters to the console. The **VioWrtTTy** function takes three parameters, which are pushed in this order: the address of the first character in the string, a word value containing the length of the string, and a word value that is the handle that identifies the screen. In this case the handle is 0.

When you pass the address of an object on the stack, you push the segment selector (which will almost always be the **DS** register) first and then the offset of the object. In a high-level language like C, this is done automatically. However, if you are using assembly language you will have to do it explicitly.

This assembly language program writes the string "Hello OS/2 World" to the console:

```
; This program writes the string "Hello OS/2 World" on
; the screen using the VIOWRTTTY API service.

PAGE ,132                          ; set page dimensions

; Set up 16-bit segments
        DOSSEG                     ; Microsoft segment conventions
        .model  SMALL              ; set model size for program
        .286
        .STACK  300H               ; set up 768 byte stack

        .DATA
MESS    DB      'Hello OS/2 World'
        .CODE
```

```
START:                                  ; beginning of code
        EXTRN   VIOWRTTTY:FAR, DOSEXIT:FAR
SPROC   PROC    FAR                     ; declare the main procedure

                                        ; push address of MESS
        PUSH    DS                      ; push segment selector
        MOV     AX, OFFSET MESS         ; get offset
        PUSH    AX                      ; push it
        PUSH    16                      ; push length of MESS
        PUSH    0                       ; handle of screen: 0
        CALL    VIOWRTTTY               ; call it

        PUSH    0                       ; setup for exit
        PUSH    0
        CALL    DOSEXIT                 ; call API exit function
SPROC   ENDP                            ; end

        END     START
```

The same program is shown here using C:

```
/*
  Write the message "Hello OS/2 World" to the screen
  using the VioWrtTTy API service.
*/

#define INCL_SUB
#include <os2.h>

char mess[17] = "Hello OS/2 World";
main()
{
  VioWrtTTy((char far*) mess, 16, 0);
}
```

Notice that the cast **char far** * is used to ensure that the pointer **mess** is passed as a FAR address. If your knowledge of C is a bit rusty, remember that the name of an array is evaluated by C to be the address of the first byte of that array. Hence **mess** is, indeed, a pointer.

Because there are many API services, the header files that contain their definitions are large and it takes the compiler a long time to read and process them. For this reason, by default the Microsoft compiler does not include all parts of the header files. Instead it uses a series of **#ifdef** statements to include many of the API service declarations conditionally. The **#ifdefs** are controlled with these symbols:

Symbol	Meaning
INCL_BASE	Include all API declarations
INCL_DOS	Include OS/2 kernel functions
INCL_SUB	Include OS/2 subsystems
INCL_DOSERRORS	Include OS/2 errors

Therefore, the symbol **INCL_SUB** is defined to have the **VioWrtTTy** declaration (which is a subsystem service) included in the program. The reason a symbol did not have to be defined in the first C program example is that some services, including **DosBeep** and **DosExit**, are always included automatically. In the chapters that follow you will learn which services require which symbol to be defined. (If you are using a non-Microsoft C compiler, you will have to determine how to include the API service declarations in your program.)

Keep in mind that a functionally similar C program can be written by using one of C's various standard library functions, such as **printf()**, instead of calling the API directly. This will be the case with many of the API services. In something as simple as the preceding program, using **printf()** would probably have been a better idea. Most of the examples in the book are in C because it provides a better means than assembly programs of presenting and illustrating the API services. Most programmers will use C to develop OS/2 applications, so it makes sense to show examples in the language that will actually be used. This means that API services that overlap parallel standard library functions will often be used to illustrate those API services. However, it may be more efficient to access an API service directly even if a similar C standard library function exists.

High-performance DOS software traditionally bypassed the C standard library functions, as well as DOS itself, in the quest for greater performance. A similar situation will exist for OS/2 programs. In several areas you will want to bypass C's standard library functions and call the API routines directly to achieve faster run-time execution. When you call a standard C function that is paralleled by OS/2, your call to the standard function is generally simply passed along to the corresponding API service. This means that two calls (one to the standard function, one to the API) are generated rather than one. When you call the API directly, however, only one call to the API service routine is generated. Since calling a routine takes time, for the fastest possible programs you should call the API directly. Keep in mind, however, that if several sections of your programs are not time critical, it makes

more sense to call the standard functions because they are more portable between operating systems and are occasionally easier to use.

THE API SERVICES

Part Two of this book covers the core API services and their use. This section will introduce the various categories of functions and the special subset of the API called the Family API (FAPI) services. The FAPI routines are the services that are common to DOS and OS/2.

The Major API Categories

The API services can be separated into five broad categories: the basic OS/2 kernel, the video subsystem, the mouse subsystem, the keyboard

DosAllocHuge	DosExit	DosGetSeg
DosAllocSeg	DosExitCritSec	DosGetShrSeg
DosAllocShrSeg	DosExitList	DosGetVersion
DosBeep	DosFileLock	DosGiveSeg
DosBufReset	DosFindClose	DosHoldSignal
DosCaseMap	DosFindFirst	DosInsMessage
DosChdir	DosFindNext	DosKillProcess
DosChgFilePtr	DosFlagProcess	DosLoadModule
DosCLIAccess	DosFreeModule	DosLockSeg
DosClose	DosFreeSeg	DosMakePipe
DosCloseQueue	DosGetCollate	DosMemAvail
DosCloseSem	DosGetCP	DosMkdir
DosCreateCSAlias	DosGetCtryInfo	DosMonClose
DosCreateQueue	DosGetDateTime	DosMonOpen
DosCreateSem	DosGetDBCSEv	DosMonRead
DosCreateThread	DosGetEnv	DosMonReg
DosCWait	DosGetHugeShift	DosMonWrite
DosDelete	DosGetInfoSeg	DosMove
DosDevConfig	DosGetMachineMode	DosMuxSemWait
DosDevIOCtl	DosGetMessage	DosNewSize
DosDupHandle	DosGetModHandle	DosOpen
DosEnterCritSec	DosGetModName	DosOpenQueue
DosErrClass	DosGetProcAddr	DosOpenSem
DosError	DosGetPrty	DosPeekQueue
DosExecPgm	DosGetResource	DosPFSActivate

Figure 2-1. The OS/2 kernel API services

DosPFSCloseUser	DosReAllocHuge	DosSetSession
DosPFSInit	DosReAllocSeg	DosSetSigHandler
DosPFSQueryAct	DosResumeThread	DosSetVec
DosPFSVerifyFont	DosRmdir	DosSetVerify
DosPhysicalDisk	DosScanEnv	DosSleep
DosPortAccess	DosSearchPath	DosStartSession
DosPTrace	DosSelectDisk	DosStopSession
DosPurgeQueue	DosSelectSession	DosSubAlloc
DosPutMessage	DosSemClear	DosSubFree
DosQCurDir	DosSemRequest	DosSubSet
DosQCurDisk	DosSemSet	DosSuspendThread
DosQFHandState	DosSemSetWait	DosSystemService
DosQFileInfo	DosSemWait	DosTimerAsync
DosQFileMode	DosSetCP	DosTimerStart
DosQFSInfo	DosSetDateTime	DosTimerStop
DosQHandType	DosSetFHandState	DosUnlockSeg
DosQueryQueue	DosSetFileInfo	DosWrite
DosQVerify	DosSetFileMode	DosWriteAsync
DosRead	DosSetFSInfo	DosWriteQueue
DosReadAsync	DosSetMaxFH	
DosReadQueue	DosSetPrty	

Figure 2-1. The OS/2 kernel API services (*continued*)

KbdCharInKbdClose	KbdPeek
KbdCustCP	KbdRegister
KbdDeRegister	KbdSetFgnd
KbdFlushBuffer	KbdSetStatus
KbdFreeFocus	KbdSetXt
KbdGetFocus	KbdShellInit
KbdGetStatus	KbdStringIn
KbdGetXt	KbdSynch
KbdOpen	KbdXlate

Figure 2-2. The keyboard subsystem services

MouClose	MouIniReal
MouDeRegister	MouOpen
MouDrawPtr	MouReadEventQue
MouFlushQue	MouRegister
MouGetDevStatus	MouRemovePtr
MouGetEventMask	MouSetDevStatus
MouGetHotKey	MouSetEventMask
MouGetNumButtons	MouSetHotKey
MouGetNumMickeys	MouSetPtrPos
MouGetNumQueEl	MouSetPtrShape
MouGetPtrPos	MouSetScaleFact
MouGetPtrShape	MouShellInit
MouGetScaleFact	MouSynch

Figure 2-3. The mouse subsystem services

VioDeRegister	VioPrtSc	VioSetCurPos
VioEndPopUp	VioPrtScToggle	VioSetCurType
VioGetAnsi	VioReadCellStr	VioSetFont
VioGetBuf	VioReadCharStr	VioSetMode
VioGetConfig	VioRegister	VioSetState
VioGetCP	VioSavReDrawUndo	VioShowBuf
VioGetCurPos	VioSavReDrawWait	VioWrtCellStr
VioGetCurType	VioScrLock	VioWrtCharStr
VioGetFont	VioScrollDn	VioWrtCharStrAtt
VioGetMode	VioScrollLf	VioWrtNAttr
VioGetPhysBuf	VioScrollRt	VioWrtNCell
VioGetState	VioScrollUp	VioWrtNChar
VioModeUndo	VioScrUnlock	VioWrtTTy
VioModeWait	VioSetAnsi	
VioPopUp	VioSetCP	

Figure 2-4. The video subsystem services

subsystem, and the Presentation Manager services. The second part of this book covers the non-Presentation Manager API services; the third part introduces the Presentation Manager. The reason for this is simple: The non-Presentation Manager services represent the core OS/2 functions. You cannot write programs that effectively use the Presentation Manager services until you understand the fundamental OS/2 routines.

There are 225 API services, not counting the Presentation Manager routines. These services are shown in Figures 2-1 through 2-4. All the API function names should be considered reserved and not used for any other purpose by your program.

DosBeep	DosQFileMode	VioGetMode
DosChdir	DosQVerify	VioGetPhysBuf
DosChgFilePtr	DosRead	VioReadCellStr
DosClose	DosRmdir	VioReadCharStr
DosDelete	DosSelectDisk	VioScrLock
DosDevConfig	DosSetFHandState	VioScrUnLock
DosDevIOCtl	DosSetFSInfo	VioScrollDn
DosDupHandle	DosSetFileInfo	VioScrollLf
DosError	DosSetFileMode	VioScrollRt
DosFileLocks	DosSetVec	VioScrollUp
DosFindClose	DosSetVerify	VioSetCurPos
DosFindFirst	DosWrite	VioSetCurType
DosFindNext	KbdCharIn	VioSetMode
DosMkdir	KbdFlushBuffer	VioShowBuf
DosMove	KbdGetStatus	VioWrtCellStr
DosNewSize	KbdPeek	VioWrtCharStr
DosOpen	KbdRegister	VioWrtCharStrAtt
DosQCurDir	KbdSetStatus	VioWrtNAttr
DosQCurDisk	KbdStringIn	VioWrtNCell
DosQFHandState	VioGetBuf	VioWrtNChar
DosQFSInfo	VioGetCurPos	VioWrtTTy
DosQFileInfo	VioGetCurType	

Figure 2-5. The API services

The Family API Services

To enable the writing of programs that will run under both DOS and OS/2, Microsoft has identified 65 API services that are applicable to both environments. These services are called the *Family API*, or *FAPI* for short. If your program uses only these services, you can run the same program under both DOS and OS/2. The FAPI functions are shown in Figure 2-5.

API SERVICE DESCRIPTION CONVENTIONS

The proper way to call an API service is shown using C function prototype notation. In fact, from a C program, the API services look like any other C library function. For example, using C prototype declarations, the proper way to call the **DosBeep** function is

```
unsigned pascal far DosBeep(unsigned freq, unsigned duration)
```

From an assembly code point of view, this declaration tells you to push the frequency first and then the duration. If you are unfamiliar with C prototypes, refer to Appendix B.

As you will see in subsequent chapters, some of the API services require that the address to a data structure be passed. Although neither the name of the structure nor the names of the fields that comprise the structure are important to OS/2—it has no knowledge of them—they are very important to the C programmer. Because each C compiler that runs under OS/2 must declare the API services and define any structures they require, it must name the structures and the fields. The trouble is that there is no reason why two different compiler manufacturers must use the same names when describing the same structures. (Remember that the API services never "see" the names, only the data.) The question is which compiler's naming conventions you choose to follow. At the time of this writing only one C compiler is available for OS/2: Microsoft C 5.10. Hence this book is written from the point of view of the Microsoft compiler. References to structure names and fields will follow the Microsoft naming conventions by default. Your compiler may use different names, but the content of the structure will be the same. (You can define your own data

structures, but doing so will result in annoying compile-time warning messages.)

One final point: The Microsoft OS/2 header files define several new type names, using the **typedef** statement, which are used in the declarations for the API services. For example, the name **USHORT** is another name for **unsigned**. However, this book shows all type declarations in their native C base types for the sake of generality and the ability to compile programs successfully with any OS/2-compatible C compiler.

II

PROGRAMMING OS/2 API SERVICES

In this section the most important core (non-Presentation Manager) OS/2 API services are discussed. Several example programs are included in each chapter. A solid understanding of the core services is important for several reasons, not the least of which is to provide support for such things as device monitors, interprocess communication, and dynamic link libraries.

3

THE SCREEN OUTPUT SERVICES

Since it is rare to create a useful program that does not display information on the screen, it seems logical to begin your tour of the OS/2 API services with those that relate to the screen. These services are commonly called the *Video I/O subsystem* (VIO for short) and are used to display text on the screen and to control the screen environment. (Graphics output is handled by the Presentation Manager services.) The names of all the functions in this subsystem begin with the prefix *Vio*. Table 3-1 lists the 43 VIO system services and gives a short description of each. This chapter covers the most important and commonly needed of these screen functions.

You might be wondering at this point why something as conceptually simple as writing output to the screen requires so many different services. Part of the answer is that OS/2 gives you a wide range of options and approaches for writing to the screen. In addition, because OS/2 is a multitasking system, it needs some VIO services to demand or control access to the screen.

As you know, the C standard library contains several functions that perform console output, including **printf()**. For the most part, when your program is performing "generic" screen output, it is easier to use these standard library functions than to call a VIO service. However, the VIO services allow significantly greater flexibility in the way text is written to the screen, including such things as displaying text in color and positioning the cursor. When your program needs to display output in a special way, you will want to use the VIO services. And, of course, several VIO functions are not paralleled by the C standard library.

Note: It is possible to bypass OS/2's built-in screen services and access the video hardware directly. The direct control of the video hardware is not only quite complicated but also seldom necessary or even desirable. The direct video hardware accessing capabilities were included in OS/2 because OS/2 has to be "all things to all programmers." But their use is not recommended for the vast majority of programming applications. This chapter deals with the screen services you will use for most (if not all) of your programming, not with the ones that allow you to manipulate the video hardware directly.

Table 3-1. The Video Subsystem Services

Service	Function
VioDeRegister	Deactivates an alternate set of VIO services
VioEndPopUp	Releases control of the screen at the end of a VioPopUp
VioGetAnsi	Returns the status of the ANSI flag
VioGetBuf	Returns the address of the logical video buffer
VioGetConfig	Returns the configuration of the video hardware components
VioGetCP	Returns the current code page
VioGetCurPos	Returns the coordinates of the cursor
VioGetCurType	Returns the dimensions of the cursor
VioGetFont	Returns the current font or font table
VioGetMode	Returns the current video mode
VioGetPhysBuf	Returns a selector to a video display buffer
VioGetState	Returns information about the current video settings
VioModeUndo	Cancels a VioModeWait
VioModeWait	Tells graphics applications when to restore its video mode
VioPopUp	Requests control of the screen
VioPrtSc	Prints the screen
VioPrtScToggle	Toggles continuous screen printing
VioReadCellStr	Reads character and attribute information from the screen
VioReadCharStr	Reads characters from the screen
VioRegister	Activates an alternate set of VIO services
VioSavReDrawUndo	Cancels a VioSavReDrawWait

Table 3-1. The Video Subsystem Services (*continued*)

VioSavReDrawWait	Notifies a process when it is necessary to save or restore the screen
VioScrLock	Prevents other processes from using the screen
VioScrollDn	Scrolls part of the screen down
VioScrollLf	Scrolls part of the screen left
VioScrollRt	Scrolls part of the screen right
VioScrollUp	Scrolls part of the screen up
VioScrUnlock	Unlocks the screen
VioSetAnsi	Sets the status of the ANSI flag
VioSetCP	Sets the current code page
VioSetCurPos	Positions the cursor at the specified coordinates
VioSetCurType	Sets the cursor dimensions
VioSetFont	Sets the current font
VioSetMode	Sets the video mode
VioSetState	Sets various display parameters
VioShowBuf	Displays the logical video buffer
VioWrtCellStr	Writes character and attribute information to the screen
VioWrtCharStr	Writes characters to the screen
VioWrtCharStrAtt	Writes characters and attributes to the screen
VioWrtNAttr	Writes attributes to the screen
VioWrtNCell	Writes the same character and attribute to the screen
VioWrtNChar	Writes the same character to the screen
VioWrtTTy	Writes a string to the screen

VIDEO ADAPTERS AND MODES OF OPERATION

Before approaching the screen services, you need to understand the various ways in which the video display hardware can function. Several different types of video adapters are currently available for the PC line of computers. The most common are the Monochrome Adapter, the CGA (Color/Graphics Adapter), PCjr, and the EGA (Enhanced Graphics Adapter). The PS/2 line of computers introduced the VGA (Video Graphics Array) adapter. Together these adapters support 19 different modes of video operation. The current video mode determines how

information is displayed on the screen. These video modes are synopsized in Table 3-2. As you can see by looking at the table, some modes are for text and some are for graphics. In a text mode only text can be displayed. The smallest user-addressable part of the screen in a text mode is one character. The smallest user-addressable part of the screen in a graphics mode is one pel. (A *pel* is the smallest individually accessible unit for a given graphics mode.)

In all PC, AT, and PS/2 computers, the display hardware uses a memory-mapped approach to displaying text. In this method a region of memory is reserved for the screen's use, and whatever this memory contains is shown on the screen. This region of memory is commonly called the *video buffer* or the *video RAM*. Exactly how this memory is organized and where it is physically located depends in part on the current video mode. However, unless you decide to bypass OS/2's screen services in favor of direct memory access of the video RAM, you will not need to worry about where the video buffer is located. Since the screen API services work only in text mode, the organization of the video RAM is always the same.

Table 3-2. The Video Modes for the Various IBM Video Adapters

Mode	Type	Dimensions	Adapters
0	Text, b/w	40×25	CGA, EGA, VGA
1	Text, 16 colors	40×25	CGA, EGA, VGA
2	Text, b/w	80×25	CGA, EGA, VGA
3	Text, 16 colors	80×25	CGA, EGA, VGA
4	Graphics, 4 colors	320×200	CGA, EGA, VGA
5	Graphics, 4 gray tones	320×200	CGA, EGA, VGA
6	Graphics, b/w	640×200	CGA, EGA, VGA
7	Text, b/w	80×25	Monochrome
8	Graphics, 16 colors	160×200	PCjr
9	Graphics, 16 colors	320×200	PCjr
13	Graphics, 16 colors	320×200	EGA, VGA
14	Graphics, 16 colors	640×200	EGA, VGA
15	Graphics, 2 colors	640×350	EGA, VGA
16	Graphics, 16 colors	640×350	EGA, VGA
17	Graphics, 2 colors	640×480	VGA
18	Graphics, 16 colors	640×480	VGA
19	Graphics, 256 colors	620×200	VGA

The Attribute Byte in Text Mode

In all text modes each character displayed on the screen is associated with an attribute byte that defines the way the character is displayed. The attribute byte associated with each character determines the color of the character, the background color, the intensity of the character, and whether it is blinking or nonblinking. The attribute byte is organized as shown in Table 3-3.

Bits 0, 1, and 2 of the attribute byte determine the foreground color component of the character associated with the attribute. For example, setting bit 0 causes the character to appear in blue. If all bits are off, the character is not displayed. Keep in mind that the colors are additive. When all three bits are on, the character is displayed in white. If you set two of the bits, either magenta or cyan is produced. The same applies to the background colors. When bits 4 through 6 are off, the background is black. Otherwise the background appears in the color specified.

The attribute value for normal text is 7, the combination of blue, green, and red. For reverse video, the value of the attribute is 70H, the combination of background blue, green, and red.

In the early days of microcomputers, the default operation of the video system displayed characters in full intensity, and you had the option to display in low intensity. However, when the IBM PC was released, it worked the other way around. The default video operation of the PC line is in "normal" intensity and you have the option to dis-

Table 3-3. The Attribute Byte Organization

Bit	Meaning When Set
0	Foreground blue
1	Foreground green
2	Foreground red
3	High intensity
4	Background blue
5	Background green
6	Background red
7	Blinking character

play characters in high intensity by setting the high-intensity bit. Finally, you can cause the character to blink by setting bit 7.

As you will see, many of the VIO functions that actually output characters to the screen also manipulate the attribute byte.

SCREEN VIRTUALIZATION AND LOGICAL VIDEO BUFFERS

In the OS/2 environment, the screen is *virtualized*. When you call a VIO service that writes output to the screen, the information you want displayed does not get put directly into the video RAM. Instead it is written to a *logical video buffer* (LVB), which is owned by the process that performs the call to a VIO service. When the process is in the foreground, OS/2 automatically maps the contents of the LVB into the physical video buffer. However, when the process is in the background, output is simply held in the LVB until that process has access to the screen. In this way OS/2 is able to prevent background processes from writing to the screen when they are not supposed to.

Each process is assigned an LVB when it begins executing. Each buffer is separate from other LVBs in the system. Since the VIO screen services apply only to text mode operation, the LVB is applicable only to text mode. If you want to use a graphics mode, you should use the Presentation Manager routines. Keep in mind that the LVB is structurally equivalent to the physical video buffer.

In the rest of this book, when a function is said to "write to the screen," remember that in most cases it is technically writing to its LVB.

THE VIDEO BUFFER ORGANIZATION

The text screen video buffer (either physical or logical) is organized in pairs of bytes. The even-numbered bytes hold the character information, and the odd-numbered bytes hold the attribute values.

VIO HANDLES

Each VIO service requires that the number of the device it is to operate on be passed. This number is called a *handle*. As OS/2 is currently implemented, all VIO device handles must be 0. However, the handle

must be passed to all VIO routines to allow for possible future enhancements of OS/2.

Note: In OS/2 many handles are represented by 16-bit quantities. For most OS/2-compatible C compilers at this time, an unsigned integer is 16 bits wide. However, it is conceivable that future C compilers specifically designed for the 80386 processor will use 32-bit unsigned integers. For this reason you may want to declare all handles as *unsigned short* (as is done in this book), which will ensure that a 16-bit integer is generated. (It is remotely possible that your C compiler will generate an 8-bit variable when the **short** modifier is used. In that case you would not want to use the **short** modifier, so check your user manual. If your compiler complies with the draft ANSI standard, you will have no trouble.)

THE VioWrtTTy SERVICE

By far the simplest, if least powerful, VIO service is **VioWrtTTy**, which outputs a string to the screen at the current cursor position. You saw a brief example of it in Chapter 2. Let's look at it in some detail here. The prototype for **VioWrtTTy** is

```
unsigned VioWrtTTy(char far * str, unsigned len, unsigned short
  handle);
```

where *str* is a pointer to the first byte of the string to be displayed, *len* is the length of the string, and *handle* is the device handle for the screen, which must be zero.

VioWrtTTy writes the specified string beginning at the current cursor position and positions the cursor after the last character written. (**VioWrtTTy** is the only screen function that updates the cursor position.) It recognizes such things as carriage returns, linefeeds, and tabs. Keep in mind that when you want **VioWrtTTy** to perform a carriage return-linefeed operation you must imbed those characters into the string. You cannot simply use the newline character as you would if you were using **printf( )**, for example. The following program illustrates how to use **VioWrtTTy**:

```
/* Demonstration of VioWrtTTy. */

#define INCL_SUB

#include <os2.h>

main()
{
  char s[80];

  strcpy(s, "this is a ");
  VioWrtTTy((char far *) s, strlen(s), 0);

  strcpy(s, "test\n");
  VioWrtTTy((char far *) s, strlen(s), 0);

  strcpy(s, "this is a second test\r\n");
  VioWrtTTy((char far *) s, strlen(s), 0);

  strcpy(s, "this is a third test\r\n");
  VioWrtTTy((char far *) s, strlen(s), 0);

}
```

The output from this program will look like this:

```
this is a test
          this is a second test
this is a third test
```

If you are programming in C, you will probably want to use **printf()** rather than **VioWrtTTy** because it is easier to use and **VioWrtTTy** does not add any greater functionality. However, there may be an exception to this rule, as described in the next section.

Note: As was mentioned in Chapter 2, because of the way the Microsoft version 5.10 C compiler has organized its OS/2 header files, you must define **INCL_SUB** at the start of each program for the VIO subsystem prototypes and types to be read into your program. If you are using a different type of C compiler, this statement may not be necessary.

VIO SERVICES VERSUS I/O REDIRECTION

You must understand one very important point about the VIO services: The output from them cannot be redirected. For example, if the program in the preceding section is called TEST, this command line will

not function as expected.

```
TEST >OUT
```

The file OUT will be created, but it will contain nothing. The program still writes its output to the screen. If you want to create redirectable output, you must use a file system service such as **DosWrite**, which will be described in Chapter 6.

THE VIO SCREEN OUTPUT SERVICES

Aside from **VioWrtTTy** there are six VIO services that write text to the screen. Let's take a look at each of them.

VioWrtCellStr

Although the VIO screen services are not hierarchical, **VioWrtCellStr** can be thought of as the lowest-level screen output function. **VioWrtCellStr** writes a string of character-attribute pairs to the screen at the specified location. OS/2 refers to a character-attribute pair as a *cell*. Keep in mind that a string of cells is not the equivalent of a C character string. A cell string is not null terminated, for example. A cell string is illustrated in Figure 3-1.

Note: Remember that when the video hardware is in text mode, the video display buffer (and each process's logical video buffer) is organized in the same fashion as the cell string: Even-numbered addresses hold the character, odd-numbered addresses contain the attribute.

The prototype for **VioWrtCellStr** is

```
unsigned VioWrtCellStr(char far * cell_str, unsigned len, unsigned
  row, unsigned col, unsigned short handle);
```

where *cell_str* is the array of cells to be displayed, and *len* is the length of the string in bytes. The parameters *row* and *col* specify the row and column coordinates of the location at which the string will be written. The *handle* parameter must be 0.

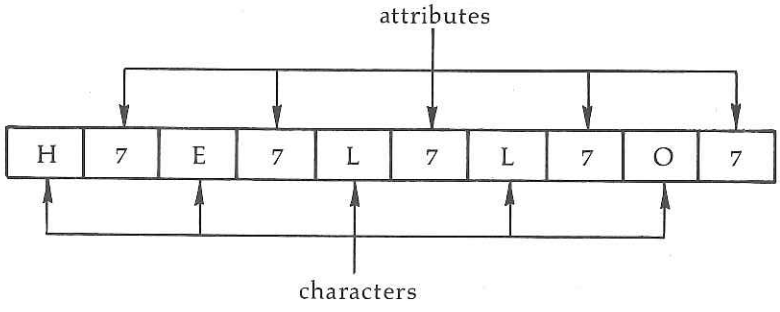


Figure 3-1. A cell string containing the word HELLO using normal video attributes

The sample program shown here outputs a very small cell string that contains the letter *A*. It is displayed first in normal video and then in reverse video.

```
/* This program demonstrates the VioWrtCellStr. */

#define INCL_BASE
#include <os2.h>

main()
{
  char c[4];

  /* write an A in normal and reverse video */
  c[0] = 'A';
  c[1] = 7;  /* normal video */
  c[2] = 'A';
  c[3] = 0x70;  /* reverse video */
  VioWrtCellStr(c, 4, 10, 10, 0);
}
```

It is important to understand that **VioWrtCellStr** neither cares about nor modifies the current cursor location. In fact, of all the VIO screen output services, only **VioWrtTTy** updates the cursor. The reason for this is simple. Updating the cursor takes time. Since the API services are designed to be as fast as possible, the designers of OS/2 decided to decouple the cursor position from most of the output routines and let the programmer move the cursor about manually. This

makes a lot of sense because most applications display information over a large area of the screen but don't generally need to move the cursor very often. (One of the examples in this chapter shows how to position the cursor manually.)

If you have programmed in a DOS environment, you will really appreciate how fast the OS/2 screen services are. The DOS character screen output routines are notoriously slow. However, the OS/2 services are nearly as fast as direct hardware-accessing methods.

One final point: OS/2 does not have a VIO function that simply outputs one character (or cell). To do this, you must call **VioWrtCellStr** (or one of the other screen output services) with a string consisting of one character attribute, which, of course, works fine.

VioWrtCharStr

You will often want to display characters on the screen using the existing screen attributes. For example, the default attribute is normal video (7) and for a great many applications, this is the attribute desired. So that you may display characters using the existing display attributes, OS/2 supplies the **VioWrtCharStr** service. Its prototype is

```
unsigned VioWrtCharStr(char far * str, unsigned len, unsigned row,
  unsigned col, unsigned short handle);
```

where *str* is the string to write and *len* is the length of the string. The string is written to the location specified by *row* and *col*. The value of *handle* must be 0. Although *str* does not need to be null terminated, it can be. This means that you can call **VioWrtCharStr** using standard C strings if you like.

This short program uses **VioWrtCharStr** to write a string at location 3,5:

```
/* This program demonstrates the VioWrtCharStr. */
#define INCL_BASE
#include <os2.h>

char s[80] = "this is a test";

main()
{
  /* write a string to the screen */
  VioWrtCharStr((char far*) s, strlen(s), 3, 5, 0);
}
```

VioWrtCharStrAtt

You often need to output text with a constant attribute. For example, you might want to prompt a user in blue text or show negative numbers in red. To accomplish this task OS/2 includes **VioWrtCharStrAtt**, which works just like **VioWrtCharStr** except that it allows you to specify a common attribute. Its prototype is

```
unsigned VioWrtCharStrAtt(char far *str, unsigned len, unsigned
  row, unsigned col, char far *attr, unsigned short handle);
```

where *str* is the string to write and *len* is the length of the string. The string is written to the location specified by *row* and *col*. The value pointed to by *attr* is the attribute that will be associated with each character in the string. The value of *handle* must be 0.

The following sample program displays its message in red letters on a blue background. The number 20 is derived from Table 3-3. (Red uses bit 2, which is the decimal value 4; blue background uses bit 4, which is the decimal value 16.)

```
/* This program demonstrates the VioWrtCharStrAtt. */

#define INCL_BASE
#include <os2.h>

main()
{
  char s[80];
  char attr;

  strcpy(s, "this will be printed in red on blue");

  attr = 20;
  VioWrtCharStrAtt((char far *) s, strlen(s), 10, 0,
                   (char far *) &attr, 0);
}
```

VioWrtNCell, VioWrtNChar, and VioWrtNAttr

Sometimes you want to write the same cell, character, or attribute several times. OS/2 uses **VioWrtNCell**, **VioWrtNChar**, and **VioWrtNAttr** to accomplish these types of operations. Their prototypes are

```
unsigned VioWrtNCell(char far *cell, unsigned count, unsigned row,
  unsigned col, unsigned short handle);
```

unsigned VioWrtNChar (char far *ch, unsigned count, unsigned row, unsigned col, unsigned short handle);

unsigned VioWrtNAttr(char far *attr, unsigned count, unsigned row, unsigned col, unsigned short handle);

Both *ch* and *attr* are byte values, but *cell* is a 2-byte character-attribute combination. In each case, *count* is the number of times the cell, character, or attribute is to be written, beginning at location *row, col.* As always, *handle* must be 0.

This program demonstrates how to call these services. The **VioWrtNCell** call writes ten *Q*s in reverse video. The **VioWrtNChar** call writes ten #s using the existing video attribute. Finally, **VioWrtNAttr** changes the attribute of ten characters to reverse video.

```
/* This program demonstrates the VioWrtNCell,
   VioWrtNChar, and VioWrtNAttr.
*/

#define INCL_BASE
#include <os2.h>

main()
{
  char cell[2];
  char attr;
  char ch;


  attr = 0x70;  /* reverse video */
  ch = '#';
  cell[0] = 'Q';
  cell[1] = 1;  /* blue */

  VioWrtNCell((char far *) cell, 10, 10, 0, 0);
  VioWrtNChar((char far *) &ch, 10, 11, 0, 0);
  VioWrtNAttr((char far *) &attr, 10, 12, 0, 0);
}
```

CURSOR POSITIONING

OS/2 supplies two services that operate on the cursor. The first, called **VioSetCurPos**, is used to set the current cursor location. The second, called **VioGetCurPos**, is used to return the coordinates of the current cursor position. Their prototypes are

unsigned VioSetCurPos(unsigned row, unsigned col, unsigned short handle);

unsigned VioGetCurPos(unsigned far *row, unsigned far *col, unsigned short handle);

For **VioSetCurPos**, the parameters *row* and *col* specify the location of the cursor. For **VioGetCurPos**, *row* and *col* are pointers to variables that will contain the current location of the cursor when the call returns. In both cases *handle* must be 0.

This short program demonstrates **VioSetCurPos** and **VioGetCurPos**. It first prints diagonal Xs across the screen and then reports the cursor's final position.

```
/* This program demonstrates the cursor position service. */

#define INCL_SUB

#include <os2.h>

main()
{
  unsigned i, j;

  /* print some Xs diagonally on the screen */
  for(i=0; i<24; i++)
    for(j=0; j<50; j+=5) {
      VioSetCurPos(i, j+i, 0);
      printf("%c", 'X');
    }

  /* now, display coordinates of the final cursor position */
  VioGetCurPos((unsigned far *) &i, (unsigned far *) &j, 0);
  printf("\ncursor is at: %d, %d\n", i, j);

}
```

SCREEN SCROLLING FUNCTIONS

OS/2 provides four services that let you scroll all or part of the screen. **VioScrollDn** scrolls the screen down, **VioScrollLf** scrolls the screen left, **VioScrollRt** scrolls the screen right, and **VioScrollUp** scrolls the screen up. The prototypes for these services are

unsigned VioScrollDn(unsigned toprow, unsigned leftcol, unsigned bottomrow, unsigned rightcol, unsigned num, char far *cell, unsigned short handle);

unsigned VioScrollLf(unsigned toprow, unsigned leftcol, unsigned bottomrow, unsigned rightcol, unsigned num, char far *cell, unsigned short handle);

unsigned VioScrollRt(unsigned toprow, unsigned leftcol, unsigned bottomrow, unsigned rightcol, unsigned num, char far *cell, unsigned short handle);

unsigned VioScrollUp(unsigned toprow, unsigned leftcol, unsigned bottomrow, unsigned rightcol, unsigned num, char far *cell, unsigned short handle);

Here, the rectangle to be scrolled is defined by *toprow, leftcol* and *bottomrow, rightcol*. The number of lines (in up and down scrolling) or spaces (in left and right scrolling), is specified by *num*. The character and attribute of the space that is scrolled in are specified by *cell*. You generally want this to be a normal video space character. Finally, *handle* must be 0.

You can scroll the entire screen by specifying the upper left and lower right coordinates of the screen. Scrolling only a portion of the screen leaves untouched the information outside the scrolled area.

You can clear the specified area by calling one of the scrolling functions with *num* having a value of -1.

This sample program demonstrates all the scrolling services. Notice that the function **clrscr()** has been created using **VioScrollUp**. The program begins by filling the screen with the uppercase alphabet and digits. It then scrolls the entire screen to the right two places. Next it scrolls a portion of the screen to the left each time you press a key until you press Q. This causes the program to scroll the area to the right with each key press. Each time you press Q, a different direction is used until all four have been tried. Sample output from this program is shown in Figure 3-2.

```
/* This program demonstrates the scrolling services. */
#define INCL_BASE
#include <os2.h>

void clrscr(void);

char s[80] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

main()
{
  register unsigned i;
  char space[2], ch;
```

```
  clrscr();

  /* Put some stuff on the screen. */
  for(i=0; i<25; i++) {
    VioWrtCharStr((char far*) s, strlen(s), i, 0, 0);
    VioWrtCharStr((char far*) s, strlen(s), i, strlen(s), 0);
  }


  /* Scroll the entire screen right two spaces, filling
     with spaces.
  */
  space[0] = ' ';
  space[1] = 7;  /* normal char */
  VioScrollRt(0, 0, 24, 79, 2, (char far *) space, 0);

  /* Now, scroll a part of the screen to the left each time
     a key is pressed.  Press 'q' to quit.
  */
  do {
    VioScrollLf(10, 10, 15, 40, 1, (char far *) space, 0);
    ch = getch();
  } while(ch!='q');

  /* Now, scroll that part of the screen back to the right
     with each keypress.  Press 'q' to quit.
  */
  do {
    VioScrollRt(10, 10, 15, 40, 1, (char far *) space, 0);
    ch = getch();
  } while(ch!='q');

  /* Now, scroll that part of the screen up
     with each keypress.  Press 'q' to quit.
  */
  do {
    VioScrollUp(10, 10, 15, 40, 1, (char far *) space, 0);
    ch = getch();
  } while(ch!='q');

  /* Now, scroll that part of the screen down
     with each keypress.  Press 'q' to quit.
  */
  do {
    VioScrollDn(10, 10, 15, 40, 1, (char far *) space, 0);
    ch = getch();
  } while(ch!='q');
}


/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGH                               DEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGH                               DEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGH    QRSTUVWXYZ0123456789ABC    DEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGH    QRSTUVWXYZ0123456789ABC    DEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGH    QRSTUVWXYZ0123456789ABC    DEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGH                               DEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

Figure 3-2. Sample output from the scrolling program

EXAMINING AND CHANGING THE VIDEO MODE

Up to this point, you have been using the default text mode of the system. In fact, the preceding sample programs simply assumed that the computer was currently using an 80 by 25 text mode. However, in actual programming situations you will probably want to know precisely what video mode is being used or to have your program explicitly set the mode it requires. To satisfy these operations OS/2 supplies the **VioGetMode** and **VioSetMode** services. Their prototypes are

```
unsigned VioGetMode(VIOMODEINFO far *data, unsigned short
  handle);
unsigned VioSetMode(VIOMODEINFO far *data, unsigned short
  handle);
```

Here *data* is a pointer to a structure that contains information about the

video mode. **VIOMODEINFO** is the name of the structure type defined by Microsoft, as shown here:

```
typedef    struct_VIOMODEINFO {
           unsigned cb;                /* number of bytes in struct */
           unsigned char fbType;       /* base type */
           unsigned char color;        /* number of colors */
           unsigned col;               /* number of columns */
           unsigned row;               /* number of rows */
           unsigned hres;              /* horizontal resolution */
           unsigned vres;              /* vertical resolution */
           unsigned char fmt_ID;       /* format ID */
           unsigned char attrib;
}VIOMODEINFO;
```

The **cb** field holds the length of the structure and is passed to both **VioGetMode** and **VioSetMode**. The rest of the fields are set by your program when **VioSetMode** is called or are set by OS/2 when **VioGetMode** is called. Let's look at these now.

The **FbType** field is a bit-map of which only the first three bits are of interest (although later versions of OS/2 can expand this). The bits are encoded as shown here:

Bit	Value	Meaning
0	0	A monochrome adapter is installed.
0	1	A color adapter is installed.
1	0	The system is in a text mode.
1	1	The system is in a graphics mode.
2	0	Color burst is enabled.
2	1	Color burst is disabled.

The **color** field contains the number of colors supported by the current (or requested) mode. This number is specified as a power of two. For example, if **color** contains a 1, two colors are supported. If it contains a 4, 16 colors are supported.

The fields **row** and **col** are used to set or return the number of text rows and columns supported.

The fields **hres** and **vres** are used to set or return the horizontal and vertical resolution (in pels).

Note: Remember that the structure type name and field names are shown using the Microsoft naming conventions. Your compiler may use different names.

The most important thing to understand about video modes in OS/2 is that they have been virtualized by the system. That is, there is no concept of requesting mode 3, for example, by using a BIOS call. (In fact, the BIOS call is not usable by OS/2.) Instead, to set the video mode you load the information about the desired mode into the **VIOMODEINFO** structure and call **VioSetMode**. If OS/2 can set the screen to the mode you desire, **VioSetMode** does so and returns 0. Otherwise it returns an error message. Approaching the video mode in this way means that your program does not have to deal with mode numbers. You need think only in terms of general descriptions.

The following program displays the status of the current video mode, sets the screen to 43-line mode, and finally displays the status of the 43-line mode. (This program requires an EGA or VGA adapter.)

```
/* This program displays the current screen mode and
   sets the screen to 43-line mode.
*/

#define INCL_SUB

void showmode(void), setmode(void);

#include <os2.h>

main()
{
  showmode();
  setmode();
  showmode();
}

void showmode()
{
  VIOMODEINFO m;

  m.cb = sizeof m;  /* must pass size of struct */
  VioGetMode((VIOMODEINFO far *) &m, 0);
  m.fbType & 1 ? printf("graphics adapter\n");
                 printf("monochrome adapter\n");
  m.fbType & 2 ? printf("graphics mode\n") :
                 printf("text mode\n");
  m.fbType & 4 ? printf("no color burst\n") :
                 printf("color burst\n");
  printf("%d colors\n", m.color);
  printf("%d columns  %d rows\n", m.col, m.row);
  printf("%d h-res  %d v-res\n\n", m.hres, m.vres);

}

void setmode()
{
  VIOMODEINFO m;

  m.cb = sizeof m;  /* must pass size of struct */
```

```
  /* get the current mode setting */
  VioGetMode((VIOMODEINFO far *) &m, 0);

  /* now, set the mode to 43 lines */
  m.fbType = 1;
  m.row = 43;
  m.hres = 640;
  m.vres = 350;
  if(VioSetMode((VIOMODEINFO far *) &m, 0))
    printf("Incompatible mode change attempted.\n");
}
```

REQUESTING VIDEO ADAPTER CHARACTERISTICS

Because of OS/2's virtual screen interface your program will rarely need to know precisely what type of video hardware is actually installed in the system. Should the need arise, however, OS/2 provides **VioGetConfig**, which returns the hardware configuration of the video adapter. Its prototype is

```
unsigned VioGetConfig(unsigned R, VIOCONFIGINFO far *data,
  unsigned short handle);
```

where *R* is reserved and must be 0. The parameter *data* is a pointer to a structure of type **VIOCONFIGINFO**, which holds the video adapter configuration when **VioGetConfig** returns. The **VIOCONFIGINFO** structure type is defined as

```
typedef  struct __VIOCONFIGINFO {
         unsigned cb;          /* size of structure */
         unsigned adapter;     /* adapter type */
         unsigned display;     /* display type */
         unsigned cbMemory;    /* size of adapter video RAM */
} VIOCONFIGINFO;
```

The **cb** variable must be loaded with the size of the structure prior to the call to **VioGetConfig**. The **adapter** variable holds the type of video adapter upon return. It is encoded like this:

Adapter Value	Adapter Type
0	Monochrome
1	CGA
2	EGA
3	VGA
7	8514A

Upon return from the service the **display** variable holds a code indicating the type of monitor attached to the system. It is encoded like this:

Display Value	Display Type
0	Monochrome
1	Color
2	Enhanced color
3	PS/2 8503 monochrome
4	PS/2 8513 color
5	PS/2 8514 color

Finally, the **cbMemory** variable holds the number of bytes of RAM available on the video adapter.

The *handle* parameter must be 0.

The following program displays the video hardware configuration of your system:

```
/* Display the video display hardware configuration. */

#define INCL_SUB

#include <os2.h>

main()
{
  VIOCONFIGINFO c;

  c.cb = sizeof c;

  VioGetConfig(0, (VIOCONFIGINFO far *) &c, 0);

  switch(c.adapter) {
    case 0: printf("Monochrome ");
      break;
    case 1: printf("CGA ");
      break;
    case 2: printf("EGA ");
      break;
    case 3: printf("VGA ");
      break;
```

```
    case 7: printf("8514A ");
  }
  printf("adapter\n");

  switch(c.display) {
    case 0: printf("Monochrome ");
      break;
    case 1: printf("Color ");
      break;
    case 2: printf("Enhanced color ");
      break;
    case 3: printf("PS/2 8503 monochrome ");
      break;
    case 4: printf("PS/2 8513 color ");
      break;
    case 5: printf("PS/2 8514 color ");
  }
  printf("display\n");

  printf("%lu bytes of memory on video adapter\n", c.cbMemory);
}
```

READING CHARACTERS FROM THE SCREEN

Because the screen is memory mapped, it is possible to read information from it. OS/2 provides two services for this purpose: **VioReadCellStr** and **VioReadCharStr**. The prototypes of these functions are

```
unsigned VioReadCellStr(char far *cellstr, unsigned far *length,
  unsigned row, unsigned col, unsigned short handle);

unsigned VioReadCharStr(char far *str, unsigned far *length,
  unsigned row, unsigned col, unsigned short handle);
```

VioReadCellStr reads both character and attribute information from the screen. **VioReadCharStr** reads only characters. Here *cellstr* is a pointer to an array that holds the character and attribute information. The *str* parameter is a pointer to a string that holds characters only. For both functions, *length* is the length of the buffer in bytes. (Remember that cells require two bytes per entry.) The location of the screen from which these services begin reading information is specified by *row* and *col*, and, as always, *handle* must be 0.

Information is read from left to right and top to bottom until the specified number of screen locations has been read. This means, for example, that you can read the entire screen using only one **VioReadCharStr** call.

This program uses **VioReadCellStr** and **VioWrtCellStr** to move what is on the top half of the screen to the bottom. It operates by copying the top half of the screen into a buffer, clearing the screen, and then copying the contents of the buffer to the bottom half of the screen. You will be surprised by how fast the VIO services accomplish their jobs.

```
/* This program uses VioReadCellStr and VioWrtCellStr to move
   what is on the top half of the screen to the bottom.
   */

#define INCL_SUB

#include <os2.h>

void clrscr();

main()
{
  unsigned size;
  char buf[2000];

  size = 1920;  /* 80 * 12 * 2 */
  if(VioReadCellStr((char far *) buf, (unsigned far *) &size,
                    0, 0, 0))
    printf("error in VioReadCellStr call\n");
  clrscr();
  VioWrtCellStr((char far *) buf, size, 12, 0, 0);
}


/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

A somewhat more useful program can be created by using **VioReadCellStr**. The program that follows saves or restores the screen to or from a disk file. It recognizes the two command line parameters: *S* and *R*. Assuming that the program is called SAVE, SAVE S saves the screen and SAVE R restores it. The contents of the screen are stored in a file called SCREEN. Notice that the program first calls **VioGetMode** to determine the dimensions of the screen so that it knows how big to make the buffer that holds the screen. Because of the virtualization of the screen interface, you can write programs that operate correctly in a

wide variety of video modes. In this case the program automatically figures out how big to make the buffer by calling **VioGetMode** and using the row and column dimensions.

```
/* This program uses VioReadCellStr and VioWrtCellStr
   to save and restore the screen.
*/

#define INCL_SUB

#include <os2.h>
#include <stdlib.h>
#include <stdio.h>

void clrscr();
void save(char *buf, unsigned size);
void restore(char *buf, unsigned size);
unsigned checkmode(void);

main(int argc, char *argv[])
{
  unsigned size;
  char *buf;

  if(argc!=2) {
    printf("usage: scr save/restore\n");
    exit(1);
  }

  size = checkmode();  /* see how big a buffer to get */

  if(!(buf = (char *) malloc(size))) {
    printf("allocation error");
    exit(1);
  }

  if(tolower(*argv[1])=='s') {
    if(VioReadCellStr((char far *) buf, (unsigned far *) &size,
                      0, 0, 0))
      printf("error in VioReadCellStr call\n");
    save(buf, size);
  }
  if(tolower(*argv[1])=='r') {
    restore(buf, size);
    clrscr();
    VioWrtCellStr((char far *) buf, size, 0, 0, 0);
  }
}

/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
```

```
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}

unsigned checkmode()
{
  struct _VIOMODEINFO m;

  m.cb = sizeof m;  /* must pass size of struct */
  VioGetMode((struct _VIOMODEINFO far *) &m, 0);
  return m.col*m.row*2;
}


/* Save the screen buffer to a disk file called SCREEN. */
void save(char *buf, unsigned size)
{
  FILE *fp;

  if(!(fp=fopen("screen", "wb"))) {
    printf("cannot open SCREEN file\n");
    exit(1);
  }
  fwrite(buf, size, 1, fp);
  fclose(fp);
}

/* Restore the previous contents of the screen. */
void restore(char *buf, unsigned size)
{
  FILE *fp;

  if(!(fp=fopen("screen", "rb"))) {
    printf("cannot open SCREEN file\n");
    exit(1);
  }

  fread(buf, size, 1, fp);
  fclose(fp);
}
```

ACCESSING THE LOGICAL VIDEO BUFFER

So far the screen services you have been using have been completely under the control of OS/2. When you performed screen output, OS/2 intercepted your output, placed it into a logical video buffer (LVB), and actually displayed the contents of the buffer on the screen as needed and allowable (which is generally what you want to happen). However, your program can directly access the logical video buffer and manually control when the buffer is displayed on the screen (assuming that the process that manipulates the LVB is in possession of the screen). Doing this still leaves OS/2 in control—just not as completely. To accomplish

these operations, OS/2 supplies the functions **VioGetBuf** and **VioShowBuf**. Their prototypes are

```
unsigned VioGetBuf(char far *bufptr, unsigned far * size, unsigned
  short handle);
unsigned VioShowBuf(unsigned offset, unsigned size, unsigned
  short handle);
```

VioGetBuf returns a far pointer, in *bufptr*, to the LVB owned by the calling process. On return the parameter *size* holds the size, in bytes, of the buffer. As always, *handle* is 0.

VioShowBuf updates the physical display buffer with the current contents of LVB beginning with the *offset* byte from the beginning of the buffer and extending for *size* bytes. The *handle* parameter must be 0.

The main (perhaps only) reason that you might want to access the LVB directly and manually control when that buffer is copied to the physical video buffer is to allow your program to construct full or partial screens in the background, perhaps using a separate thread of execution, and then very rapidly swap them into view. To see how this process works, this simple program fills the first 1000-character positions in the LVB with Xs, waits for a key press, and then displays the logical buffer. You will be amazed at how fast the screen is updated.

```
/* This program demonstrates the use of VioGetBuf and
   VioShowBuf.  These services allow you to construct a
   screen in the  background and then display it.
*/

#define INCL_SUB

#include <os2.h>

main()
{
  unsigned size, i;
  char far *p;

  /* get the address of the logical video buffer */
  VioGetBuf((unsigned long far *) &p, (unsigned far *) &size, 0);

  /* Put 1000 Xs into the LVB starting at the upper left
     corner. */
  for(i=0; i<1000; i++) {
    *p = 'X';
    p+=2; /* skip pass attribute byte */
  }
```

```
  /* wait for keypress and then show image */
  getch();
  VioShowBuf(0, size, 0);
  getch();
}
```

CURSOR AND FONTS

It is not uncommon for a program to need to know the type and size of the current text font or cursor. Sometimes it is also desirable to change the size and shape of the cursor, perhaps to signal the need for special input by the user. This section will examine some of the services that make these types of manipulations possible.

VioGetFont

OS/2 provides two services that can examine and set the current text font of the system. They are called **VioGetFont** and **VioSetFont**, respectively. While the subject of generating and using custom fonts is beyond the scope of this book, the **VioGetFont** service has several important uses that apply to a wide variety of programming situations. Its prototype is

```
unsigned VioGetFont(VIOFONTINFO font, unsigned short han-
  dle);
```

The parameter *font* points to a structure of type **VIOFONTINFO**, which holds information about the font upon return from the call. The *handle* parameter must be 0.

The structure type **VIOFONTINFO** is defined as

```
typedef  struct _VIOFONTINFO {
           unsigned cb;          /* length of structure */
           unsigned type;        /* current or ROM font? */
           unsigned cxCell;      /* number of horizontal pels in a
                                    character cell */
           unsigned cyCell;      /* number of vertical pels in a
                                    character cell */
           char far *pbData;     /* pointer to buffer which will
                                    hold the font table */
```

```
        unsigned cbData;    /* number of bytes in table */
} VIOFONTINFO;
```

If **type** is 0, the characteristics of the current RAM font are obtained. If it is 1, the information pertaining to the ROM font is returned. The fields of most interest for general use are **cxCell** and **cyCell** because they describe the dimensions of a character cell. You will soon see how to put this information to work.

VioGetCurType and VioSetCurType

As you probably know, most PCs do not have a fixed, predefined cursor. Instead the cursor is dynamically created and maintained by the operating system. In OS/2 you can examine the current size of the cursor and set its size as you desire. To examine the current dimensions of the cursor use **VioGetCurType**, whose prototype is shown here:

```
unsigned VioGetCurType(VIOCURSORINFO far *cdata, unsigned
  short handle);
```

The *handle* parameter must be 0. The *cdata* parameter points to a structure of type **VIOCURSORINFO** that will hold the current cursor information when the service returns. It is defined like this:

```
typedef  struct __VIOCURSORINFO {
         unsigned yStart;   /* top line of cursor */
         unsigned cEnd;     /* bottom line of cursor */
         unsigned cx;       /* width of cursor */
         unsigned attr;     /* cursor attribute */
} VIOCURSORINFO;
```

After the call the variable **yStart** holds the number of the line, from the top of a cell, where the cursor's top line is located. The top line in a cell is always 0, the bottom line is always $N-1$, where N is the number of vertical pels in a cell. The **cEnd** variable holds the number of the bottom line of the cursor. The **cx** variable holds the width, in columns, of

the cursor. For text modes, this value is always 1. The **attr** variable holds the cursor attributes. At the time of this writing, the value −1 means that the cursor is hidden. Any other value means that the cursor is displayed normally.

To set the size and type of a cursor, use **VioSetCurType**, which has this prototype:

```
unsigned VioSetCurType(VIOCURSORINFO far *cdata, unsigned
  short handle);
```

where the parameters have the same meaning as just described for **VioGetCurType**. To change the way the cursor looks, simply load new values into the structure variables pointed to by *cdata* and call **VioSetCurType**.

The following example program illustrates how to examine the current font and set the cursor. Inside the function **prompt()** it makes use of the fact that the size of a character cell also limits the size of the text cursor. It uses this information to construct a special, custom cursor that is a square block equidistant from the top and bottom of the cell. This function also illustrates how to update the cursor manually so that it coincides with the last write operation.

```
/* This program demonstrates how color can be used
   to highlight a prompting message using a custom cursor.
*/

#define INCL_SUB

#include <os2.h>

unsigned prompt(char *, unsigned, unsigned, unsigned char);

/* define macro names for the colors codes */
#define BLUE     1
#define GREEN    2
#define RED      4
#define INTENSE  8
#define B_BLUE   16
#define B_GREEN  32
#define B_RED    64
#define BLINK    128

main()
{
  unsigned result;

  result = prompt("Enter a Number: ", 10, 0, GREEN | BLINK | B_RED);

  printf("result is %d\n", result);
}
```

```
/* Display a prompt at the specified location using the
   specified video attribute and return an integer response.
*/
unsigned prompt(char *s, unsigned row, unsigned col,
                unsigned char attr)
{
  unsigned result;
  VIOCURSORINFO c;
  VIOFONTINFO f;
  unsigned gap;


  /* show the prompt */
  VioWrtCharStrAtt((char far *) s, strlen(s),
                   row, col, (unsigned char far *) &attr, 0);
  /* move the cursor to the appropriate location */
  VioSetCurPos(row, col+strlen(s), 0);

  /* see how tall current font is */
  f.cb = sizeof (VIOFONTINFO);
  f.type = 0;  /* get current font */
  f.pbData = (void far *) 0;
  f.cbData = 0;
  VioGetFont((VIOFONTINFO far *) &f, 0);
  /* now, compute gap */
  gap = f.cyCell / 3;

  /* make a custom cursor which is a square block that is
     situated in the middle of the cell */
  c.yStart = gap;
  c.cEnd = f.cyCell - gap;
  c.cx = 1;
  c.attr = 0;
  VioSetCurType((VIOCURSORINFO far *) &c, 0);

  scanf("%d", &result);
  return result;
}
```

VioPopUp AND VioEndPopUp

As has been stated a few times in this chapter, output is sent to the physical screen only when the process sending the output is the one shown on the screen. However, it may have occurred to you that a background process may sometimes need to access the screen for a short period of time—to report an error, for example. There must be some method by which the background process can request access to the screen. In a related situation, it is possible to detach a program from the command processor. The program then runs in background mode. However, should the detached process require the screen, some means must exist for it to demand temporary use of the screen. To

meet these needs OS/2 provides the **VioPopUp** and **VioEndPopUp** services. **VioPopUp** is used to request temporary access to the screen (and keyboard). **VioEndPopUp** releases the console and causes OS/2 to resume normal operation. Their prototypes are

```
unsigned VioPopUp(unsigned far *wait, unsigned short handle);
unsigned VioEndPopUp(unsigned short handle);
```

Here, *handle* must be 0. The value pointed to by *wait* determines what the process issuing the **VioPopUp** does if the screen is not immediately available. If bit 0 is set, the process waits until it can access the screen; if bit 0 is cleared, the process continues without access to the screen. The rest of the bits in the value pointed to by *wait* are reserved and should be set to 0.

It is important to understand that when a **VioPopUp** call is successful in gaining access to the screen, it is actually gaining access to, and is in complete control of, the screen, the keyboard, and the mouse. That is, it takes over the entire console environment.

When a background or detached process requires the console, it first calls **VioPopUp**. It then does whatever it needs to do and finishes by calling **VioEndPopUp** to return control of the console to OS/2. In principle no pop-up should dominate the screen for very long because it disrupts the normal operation of the machine.

The following program shows how a program can request the screen. It begins by sleeping for 5000 milliseconds, using the **DosSleep** command, which tells the process to suspend execution for the specified number of milliseconds. (**DosSleep** will be discussed at greater length later.) After the specified period of time, the process resumes, issues the **VioPopUp** call, prints the message "Hello—press a key," and then terminates with a call to **VioEndPopUp**. Assuming the name of the program is PU, the best way to see this program in action is to execute it as a detached process using this command line:

```
DETACH PU
```

Here is the program:

```
/* This program illustrates the VioPopUp and
   VioEndPopUp services.
*/
```

```
#define INCL_SUB
#define INCL_DOS

#include <os2.h>

main()
{
  unsigned wait;

  DosSleep(5000L);  /* sleep for a while */

  wait = 1;

  /* demand the screen and wait for it */
  VioPopUp((unsigned far *) &wait, 0);

  printf("hello - press a key");
  getch();

  /* release the screen */
  VioEndPopUp(0);
}
```

VioPopUps are exceptions to the rule when it comes to the way OS/2 handles the console. First, while a pop-up is active, the user cannot switch to another process or to an upper-level shell. The pop-up owns the screen. Second, only one pop-up can be active at any one time. A second process requesting a pop-up will be suspended. Third, when a pop-up activates, the screen is automatically placed into 80 by 25 text mode. When the pop-up ends, the screen is returned to its previous mode.

When a **VioPopUp** is active, you can use only the VIO functions shown here. (The others will not work and will return an error.)

VioEndPopUp

VioGetAnsi

VioGetCurPos

VioGetCurType

VioGetMode

VioScrollDn

VioScrollLf

VioScrollRt

VioScrollUp

VioSetCurPos

VioWrtCellStr

VioWrtCharStr

VioWrtCharStrAttr

VioWrtNAttr

VioWrtNCell

VioWrtNChar

VioWrtTTy

As you will see in a later chapter, the **VioPopUp** and **VioEndPopUp** functions are very important in OS/2 because they help support OS/2's version of the popular DOS *Terminate and Stay-Resident* utility programs.

4

THE KEYBOARD SERVICES

Because most programs written for personal computers are interactive, the API keyboard services are very important. This chapter examines several of OS/2's keyboard input routines. All the keyboard services begin with the characters **Kbd** and are sometimes referred to as the *KBD services*. The 16 keyboard services are shown and briefly described in Table 4-1.

Several of the KBD services overlap keyboard input functions provided by high-level languages such as C. For very simple, "generic" keyboard input, it is fine to use the C standard library functions. To gain full use of the keyboard, however, you will need to use the OS/2 services.

This chapter begins with a short discussion of how the keyboard generates signals and how these signals are processed. It then examines the most common keyboard services.

SCAN AND CHARACTER CODES

You might be surprised to learn that a PC keyboard does not generate the ASCII codes for the letters shown on the keys. The keyboard actually has no "knowledge" of what characters are displayed on its keys. Instead, each time a key is pressed, the keyboard generates a value that corresponds to the key's position on the keyboard. This value is called a *scan code* or, occasionally, a *position code*. The scan codes associated with each key on the IBM PS/2-compatible enhanced keyboard are shown in Figure 4-1. Notice that no scan code has the value 0.

You might be wondering why the keyboard generates scan codes instead of the actual ASCII characters that correspond to the keys. The answer is that designing a keyboard this way makes it flexible, that is,

Table 4-1. The Keyboard Services

Service	Function
KbdCharIn	Reads a character and scan code from the keyboard buffer
KbdClose	Closes the logical keyboard
KbdDeRegister	Deactivates an alternative set of keyboard services
KbdFlushBuffer	Flushes keyboard input buffer
KbdFreeFocus	Releases the keyboard
KbdGetFocus	Acquires the keyboard
KbdGetStatus	Reads keyboard status
KbdOpen	Opens a logical keyboard
KbdPeek	Examines but doesn't remove a character and scan code from the keyboard buffer
KbdRegister	Activates an alternative set of keyboard services
KbdSetFgnd	Sets foreground keyboard priority
KbdSetStatus	Sets keyboard status
KbdShellInit	Initializes the keyboard shell
KbdStringIn	Reads a string of characters from the keyboard
KbdSynch	Synchronizes access to the keyboard
KbdXlate	Translates a scan code into a character code

usable in the widest variety of situations. Many foreign languages use some characters that are different from those used by English, and the layout of the keyboard in some countries is slightly different from the U.S. version. Some people prefer a keyboard layout called the Dvorak keyboard, which is supposed to increase typing efficiency. By generating scan codes instead of ASCII character codes OS/2 can translate those codes any way it sees fit. In other words, a given scan code can be mapped onto the ASCII code equivalent required by the situation.

What Happens When You Press a Key

Each time you press a key, the keyboard generates an interrupt in the main system unit. When this happens, OS/2 temporarily stops what it is doing and executes a routine that reads the scan code of the key you pressed. Each time you release a key, another interrupt is generated and a special *release code* is sent to the system unit. The release code is

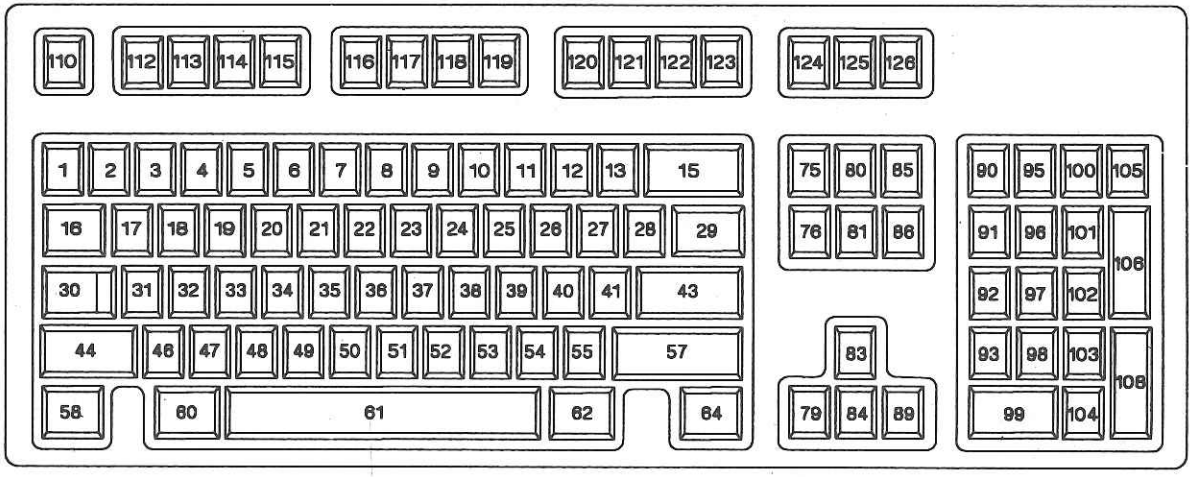


Figure 4-1. The keyboard scan codes for the PS/2-compatible enhanced keyboard

the scan code plus 128. In some computer literature the key press and release interrupts are called *make and break interrupts.*

What OS/2 Does with the Scan Code

As you now know, the keyboard sends to the system unit only a scan code, which is essentially a key position code. OS/2 contains a routine that translates this scan code into a character code. The input routine uses an ordered table of character codes that correspond to the scan codes. For example, the first entry in the table contains the character code for the ESC key, which generates a scan code of 1. The input routine searches this table to match the scan code with its proper character code. This character table is called the *character translation table.* Once the search is complete, OS/2 puts the scan code and the ASCII character code into the keyboard buffer, where it stays until your program requests keyboard information.

The PC keyboard contains several keys for which there are no ASCII character equivalents, for example, the arrow keys and the function keys. When one of these keys is looked up in the character table, its corresponding character value is 0 (or E0H in a few cases), which indicates that a non-ASCII key has been pressed. When the character code is 0, your program must examine the scan code to determine which key was pressed.

The only time the character code is E0H is when you press a key unique to the PS/2 enhanced keyboard. For example, arrow keys not on the numeric keyboard generate an E0H for their character codes although their scan codes are the same as those on the numeric keyboard. By leaving the scan codes the same, but distinguishing among them with the character codes, your software can tell them apart. The same is true of the HOME, PGUP, END, PGDN, INS, and DEL keys, which are found on both the numeric keypad and elsewhere on an enhanced keyboard.

Most high-level language keyboard input functions discard the scan code and use only the character code. This means that you generally cannot use these functions to read special keys, such as the function or arrow keys. You will find the KBD services of OS/2 particularly useful for this reason. (Remember that when your programs need only character input, it is fine to use a high-level language's standard input functions. However, for high-performance, screen-oriented programs, you will probably want to be able to recognize the various special keys.)

KEYBOARD SERIALIZATION

A program can contain multiple threads, but OS/2 cannot automatically keep the keyboard requests for separate programs from becoming mixed and confused when two or more threads in the same program make simultaneous keyboard requests. It is up to you to prevent the input for one thread from becoming mixed with input for another thread. You have to make sure that each thread requesting keyboard input has sole use of the keyboard. This means that access to the keyboard must be *serialized:* Each request for the keyboard must wait until the previous request has released it. OS/2 has several services that provide for resource serialization, but discussion of these services is deferred until later in this book when multitasking issues are discussed. (You need to understand more about OS/2 before we can develop multithread programs to illustrate the serialization concepts.)

KEYBOARD HANDLES AND LOGICAL KEYBOARDS

Each KBD routine has as one of its parameters the handle of the keyboard on which it is operating. Unlike the screen services, in which the handle was always 0 (at least for OS/2 version 1.0), the keyboard functions can operate on logical keyboards with their own keyboard buffers. This means that the keyboard routines can take handle values other than 0. These logical keyboards may be connected to the physical keyboard for only short periods of time. For example, a multithread process may have several logical keyboards sharing access to the physical keyboard. When a logical keyboard is bound to the physical keyboard, it is said to be the *focus* of the physical keyboard.

The physical keyboard is always referred to by using a handle value of 0. The examples in this chapter use this handle. If your application does not use multiple threads, you should use 0 for the keyboard handle.

COOKED VERSUS RAW KEYBOARD INPUT MODES

OS/2 supports two separate keyboard input modes. The default, and by far the most common, is called *cooked mode.* Cooked mode is essentially

ASCII mode. In this mode OS/2 recognizes the carriage return character as an end-of-line character rather than a character to be passed back from the keyboard, for example. In other words, in cooked mode OS/2 is free to perform various character translations. The opposite of cooked is *raw mode*, in which each character pressed on the keyboard is actually entered into the keyboard buffer without any modifications. You will use cooked mode for most applications.

KbdCharIn

Perhaps the most important KBD service is **KbdCharIn**, which returns the character and scan code of the last key pressed, along with some other information. You may be surprised to see how versatile this service is. Its prototype is

```
unsigned KbdCharIn(KBDKEYINFO far *key, unsigned nowait,
  unsigned handle);
```

where *key* is a pointer to a structure of type **KBDKEYINFO**, which is defined as follows:

```
typedef struct _KBDKEYINFO {
        unsigned char chChar;     /* character code */
        unsigned char chScan;     /* scan code */
        unsigned char fbStatus;   /* character status */
        unsigned char bNlsShift;  /* reserved */
        unsigned fsState;         /* shift key status */
        unsigned long time;       /* time when key pressed */
} KBDKEYINFO;
```

Upon return from the call, the **chChar** field contains the ASCII character code from the key pressed, unless it was a non-ASCII key, such as an arrow key. For special keys, this field will be 0 or E0H. The **chScan** field holds the scan code of the key.

The **fbStatus** field is encoded as follows. If bit 0 is set, the shift status is returned but no key is returned. Bits 1 through 4 are reserved. Bit 5 requests immediate conversion. Bits 6 and 7 are encoded as shown in the following table.

Bit 6	Bit 7	Meaning
0	0	Undefined
1	0	Final character of 2-byte character; no keypress still pending
0	1	Interim character (keypress)
1	1	Final character of 2-byte character; keypress still pending

Do not be confused by this table. Some foreign languages, such as Japanese, have characters large enough to require 2-byte character sets. When these sets are in use, your program needs to know whether it is reading the beginning or ending character of a 2-byte set. However, for English language use, you do not need to worry about this. For normal 1-byte character sets, the only bit that is important is 6. When bit 6 is set, a key is waiting to be read, that is, a key has been pressed. When bit 6 is cleared, no keys are waiting in the keyboard buffer.

The **fsState** field returns the states of the various shift keys. In this context the term *shift key* refers to any key that changes the state of the keyboard. The states of the shift keys are encoded into **fsState** as shown in Table 4-2.

Table 4-2. The Encoding of the Shift Key Status into the **fsState** Field

Bit	Meaning When Set
0	Right SHIFT key pressed
1	Left SHIFT key pressed
2	A CTRL key pressed
3	An ALT key pressed
4	SCROLL LOCK on
5	NUM LOCK on
6	CAPS LOCK on
7	INS on
8	Left CTRL key pressed
9	Left ALT key pressed
10	Right CTRL key pressed
11	Right ALT key pressed
12	SCROLL LOCK pressed
13	NUM LOCK pressed
14	CAPS LOCK pressed
15	SYS RQ key pressed

The value of the *nowait* parameter determines whether **KbdCharIn** waits until a key is pressed (i.e., a key is in the buffer) or returns immediately. If *nowait* is 0, **KbdCharIn** waits until a key is pressed; if *nowait* is 1, **KbdCharIn** returns without a character if none are in the keyboard buffer.

For a simple first example of **KbdCharIn**, this program waits for a keypress and displays the character plus the time (in milliseconds) at which it was pressed:

```
/* Reading a key using KbdCharIn. */

#define INCL_SUB
#include <os2.h>

main()
{
  KBDKEYINFO k;

  /* call KbdCharIn and wait for a keypress */
  KbdCharIn((KBDKEYINFO far *) &k, 0, 0);
  printf("You pressed: %c\n", k.chChar);

  /* show the time */
  printf("at %ld\n", k.time);
}
```

There are a few important points to remember about the **KbdCharIn** service:

1. It does not echo the characters to the screen. Your program will have to do this manually if echoing is desired.
2. If you are reading 2-byte character set codes, you will have to call **KbdCharIn** twice.
3. Remember that when a special key is pressed, the value of **chChar** is 0 (or E0H).

Using KbdCharIn to Check for a Keypress

As was mentioned in the description of **KbdCharIn**, you can determine if a key has been pressed by examining the **fbStatus** field of the key information structure. If bit 6 is set, a key is waiting in the keyboard buffer. If bit 6 is cleared, no key is waiting; hence, no key has been pressed. The following program is a modification of the WHOOP pro-

gram shown in Chapter 2. Here the standard C function **kbhit()** has been replaced by a call to **KbdCharIn** followed by a test on bit 6 of the **fbStatus** field. Notice that **KbdCharIn** is called with the *nowait* parameter set to 1, indicating that the service is not to wait for a keypress but to return the status information at once.

```
/* Checking for a keypress using KbdCharIn. */

#define INCL_SUB
#include <os2.h>

main()
{
  KBDKEYINFO k;
  char c[2];
  register int i;

  printf("Press any key to hear sounds.\n");

  /* wait for a keypress */
  KbdCharIn((KBDKEYINFO far *) &k, 0, 0);

  printf("Press any key to terminate.\n");
  for(;;) {
    for(i=100; i<2500; i+=50) {
      DosBeep(i, 1);
      /* don't wait for keypress */
      KbdCharIn((KBDKEYINFO far *) &k, 1, 0);
      /* see if a key has been pressed */
      if(k.fbStatus & 64) break; /* stop on keypress */
    }
    if(k.fbStatus && 64) break; /* stop if key pressed */
  }
}
```

There is another way to see if there is a key waiting to be read from the keyboard buffer.

Showing the Status of the Shift Keys

The **fsState** field of the character information structure holds the current status of the shift keys. It also indicates the state of the toggle keys: NUM LOCK, CAPS LOCK, and SCROLL LOCK. The toggle keys control internal flags that keep their related functions in one state or another, changing with each keypress. The following program displays the shift keys that are pressed and the state of the toggle keys. To use the program, press down a shift key and then strike a regular key. The program shows which shift key you pressed.

```
/* Display the status of the shift keys. */

#define INCL_SUB
#include <os2.h>

void show_shift_status(unsigned status);

main()
{
  KBDKEYINFO k;
  char c[2];

  printf("Press 'q to terminate.\n");
  for(;;) {
    /* wait for keypress */
    KbdCharIn((KBDKEYINFO far *) &k, 0, 0);

    show_shift_status(k.fsState);

    if(k.chChar=='q') break; /* stop on 'q' */
  }
}


/* Display the shift status of the keyboard. */
void show_shift_status(unsigned status)
{
  printf("\n");
  if(status & 1) printf("Right shift pressed\n");
  if(status & 2) printf("Left shift pressed\n");
  if(status & 4) printf("A control key pressed\n");
  if(status & 8) printf("An Alt key pressed \n");
  if(status & 16) printf("Scroll Lock on\n");
  if(status & 32) printf("Num Lock on\n");
  if(status & 64) printf("Caps Lock on\n");
  if(status & 128) printf("Ins pressed\n");
  if(status & 256) printf("Left Control pressed\n");
  if(status & 512) printf("Left Alt key pressed\n");
  if(status & 1024) printf("Right Control key pressed\n");
  if(status & 2048) printf("Right Alt key pressed\n");
  if(status & 4096) printf("Scroll Lock pressed\n");
  if(status & 8192) printf("Num lock pressed\n");
  if(status & 16384) printf("Caps lock pressed\n");
  if(status & 32768) printf("SysRq key pressed\n\n");
}
```

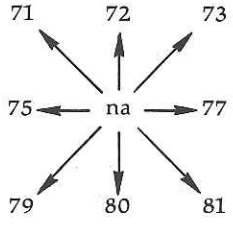
Checking for Scan Codes

If the character code returned by **KbdCharIn** is either 0 or E0H, the key pressed is not a standard ASCII key but a special key. The following program illustrates how to check for ASCII and non-ASCII keys. It waits for a keypress and prints either the character, if the key is ASCII, or its scan code if the key is non-ASCII. It also tells whether the key is unique to the enhanced keyboard.

```
/* Display character or scan code for a key. */

#define INCL_SUB
#include <os2.h>

main()
{
  KBDKEYINFO k;
  char c[2];


  /* wait for a keypress */
  KbdCharIn((KBDKEYINFO far *) &k, 0, 0);

  if(!k.chChar)
     printf("Special key; scan code is %d", k.chScan);
  else if(k.chChar==0xE0)
    printf("Enhance KB special key; scan code is %d", k.chScan);
  else printf("Key is ASCII char %c", k.chChar);
}
```

You can use this program to determine the scan codes of the special keys.

You can use the scan codes returned by the arrow keys (and their diagonal neighbors on the numeric keypad) to control the movement of the cursor in your programs. For example, you might use the arrow keys to move between menu entries. A short program in this section illustrates some of the basic concepts behind controlling the cursor with the arrow keys. The program allows you to "drive" the cursor around on the screen using the arrow keys and the keys on the diagonal of the numeric keypad. The scan codes of the keys on the keypad are shown here with the direction they will move the cursor.



The program creates and initializes two variables, **r** and **c**, which hold the current row and column coordinates of the cursor. Each time you press an arrow key, these counters are updated and the cursor is moved to its new position. Notice that out-of-range conditions are tested and corrected before the cursor is moved. Also, this program assumes that the computer is in the default text mode 80 by 25.

```
/* This program uses the arrow keys to "drive" the cursor
   around on the screen.
*/

#define INCL_SUB
#include <os2.h>

void clrscr(void);

main()
{
  KBDKEYINFO k;
  signed r, c;

  r = 0; c = 0;
  clrscr();

  VioSetCurPos(12, 30, 0);
  printf("Press 'q' to quit");

  VioSetCurPos(0, 0, 0);
  do {

    /* wait for a keypress */
    KbdCharIn((KBDKEYINFO far *) &k, 0, 0);

    switch(k.chScan) {
      case 72: r -= 1; /* up */
        break;
      case 80: r += 1; /* down */
        break;
      case 77: c += 1; /* right */
        break;
      case 75: c -= 1; /* left */
        break;
      case 71: r -= 1; /* up, left */
        c -= 1;
        break;
      case 73: r -= 1; /* up, right */
        c += 1;
        break;
      case 79: r += 1; /* down, left */
        c -= 1;
        break;
      case 81: r += 1; /* down, right */
        c += 1;
    }

    /* disallow out-of-range coordinates */
    if(c < 0) c = 0;
    if(c > 79) c = 79;
    if(r < 0) r = 0;
    if(r > 24) r = 24;

    /* move the cursor */
    VioSetCurPos(r, c, 0);

  } while(k.chChar != 'q');
}


/* A simple way to clear the screen by filling
   it with spaces.
*/
```

```
void clrscr()
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

USING KbdPeek

In the previous section you saw how **KbdCharIn** could be used to return information about the keyboard and the status of the keyboard buffer. For example, it was used to determine whether a key had been pressed. However, there is one drawback to using **KbdCharIn** to interrogate the status of the keyboard: In the process of determining the keyboard status it also reads any key waiting in the buffer. This is fine if you want that key read, but it is a problem when all you want to do is determine the status of the keyboard. To solve this problem OS/2 supplies the function **KbdPeek**, which returns the same status information as **KbdCharIn** but does not remove the character or scan code from the keyboard buffer. The prototype for **KbdPeek** is

```
unsigned KbdPeek(KBDKEYINFO far *key, unsigned handle);
```

(See the description of **KbdCharIn** for a complete description of the **KBDKEYINFO** structure.)

You can use **KbdPeek** to construct various functions that describe the state of the keyboard. One obviously useful function is called **keypress()**. It returns true if a key is waiting in the buffer and false otherwise. The **keypress()** function is shown here:

```
/* Return 1 if key pressed; 0 otherwise. */
keypress()
{
  KBDKEYINFO k;

  /* check for keypress */
  KbdPeek((KBDKEYINFO far *) &k, 0);
  return k.fbStatus & 64;
}
```

By using **keypress()** you can rewrite the WHOOP program to use this function rather than calling **KbdCharIn** to see when a key is pressed.

The new version is shown here:

```
/* Checking for a keypress using KbdPeek. */

#define INCL_SUB
#include <os2.h>

int keypress(void);

main()
{

  KBDKEYINFO k;
  register int i;

  printf("Press any key to hear sounds.\n");

  /* wait for a keypress and discard character */
  KbdCharIn((KBDKEYINFO far*) &k, 0, 0);

  printf("Press any key to terminate.\n");
  for(;;) {
    for(i=100; i<2500; i+=50) {
      DosBeep(i, 1);
      if(keypress()) break;
    }
    if(keypress()) break; /* stop if key pressed */
  }
}


/* Return 1 if keypressed; 0 otherwise. */
keypress()
{
  KBDKEYINFO k;
  /* check for keypress */

  KbdPeek((KBDKEYINFO far *) &k, 0);
  return k.fbStatus & 64;
}
```

Notice that in the **main()** function, the **KbdCharIn** service is still used to read the initial keypress. Why? The reason is that **KbdPeek** does not remove the character from the key buffer or reset the buffer in any way. Once a key is pressed (if it is not removed), repeated calls to **KbdPeek** will return that a character is pending in the keyboard buffer. Had the character not been read, the calls to **keypress()** later in **main()** would have returned true, and the program would have terminated immediately.

Although most C compilers support the more-or-less standard function **kbhit()**, the advantage of using **KbdPeek** to determine keyboard status is that it returns additional information about the state of the keyboard buffer.

CLEARING THE KEYBOARD BUFFER

Your program will sometimes want to ignore the existing contents of the keyboard buffer and start fresh. For example, an error condition may require the user to enter a response. If the error occurs in the middle of some other interactive operation, there may be characters already waiting in the keyboard buffer. For the user to respond correctly to the error condition, the contents of the keyboard buffer must be cleared. And it is sometimes a good idea to clear any characters that may be in the keyboard buffer before highly critical input is to be read to ensure that no "garbage" characters (caused by the user absent-mindedly tapping on the keyboard) are accidentally read. The act of clearing the keyboard buffer (or just about any type of buffer, for that matter) is called *flushing* the buffer. To accomplish this OS/2 provides the **KbdFlushBuffer** service. Its prototype is

```
unsigned KbdFlushBuffer(unsigned handle);
```

A call to **KbdFlushBuffer** clears the keyboard buffer and resets the appropriate status flags to indicate this fact.

For example, it is not a bad idea to clear the keyboard buffer when a program begins execution. This fragment shows how this can be done by using **KbdFlushBuffer**:

```
.
.
.
main()
{
  KbdFlushBuffer(0);
  .
  .
  .
```

USING KbdGetStatus AND KbdSetStatus

Both **KbdCharIn** and **KbdPeek** return status information about the state of the keyboard buffer. However, your program might need to know other pieces of information that are not returned by these services. For example, your program may need to operate differently when

the keyboard is in raw rather than cooked mode. To fill this need, OS/2 supplies the **KbdGetStatus** service, which returns a complete status packet. Its prototype is

```
unsigned KbdGetStatus(KBDINFO far *info, unsigned handle);
```

where *info* is a pointer to a structure of type **KBDINFO**, which is defined like this:

```
typedef  struct _KBDINFO {
         unsigned  cb;            /* size of structure */
         unsigned  fsMask;        /* modified states */
         unsigned  chTurnAround;  /* EOL char */
         unsigned  fsInterim;     /* interim char flags */
         unsigned  fsState;       /* shift key states */
} KBDINFO;
```

The **cb** field must hold the size of the structure before the call to **KbdGetStatus** is made. The **fsMask** shows the current input mode (cooked or raw) and whether keystrokes are automatically echoed to the screen. It also shows which KBD subsystem settings are to be changed by a subsequent **KbdSetStatus** call. (More on **KbdSetStatus** in a moment.) This information is encoded into **fsMask** as shown here:

Bit	Meaning When Set
0	Echo on
1	Echo off
2	Raw mode
3	Cooked mode
4	Shift state to be changed
5	Interim flags to be changed
6	Turnaround character to be changed
7	Length of turnaround character

The **chTurnAround** character is used to terminate a line of input. By default this is the carriage return character, but it can be any character. Bit 7 indicates the length of the turnaround character. If bit 7 is set, the

character is 2 bytes long. Otherwise it is 1 byte long. The rest of the bytes in this variable are reserved.

The **fsInterim** field indicates the state of the keyboard buffer as described in **KbdCharIn**. The **fsState** field holds the status of the shift keys. The bits are encoded like this:

Bit	Meaning When Set
0	Right SHIFT key pressed
1	Left SHIFT key pressed
2	CTRL key pressed
3	ALT key pressed
4	SCROLL LOCK mode on
5	NUM LOCK mode on
6	CAPS LOCK mode on
7	INS mode on
8–15	Reserved

In **KbdGetStatus** the *handle* parameter must be 0.

This program uses **KbdGetStatus** to report the status of the input mode and whether keystrokes are automatically echoed to the screen. It then reports the current shift status.

```
/* Using KbdGetStatus. */

#define INCL_SUB

#include <os2.h>

void showmask(unsigned);
void show_shift_status(unsigned);

main()
{
  KBDINFO ki;

  ki.cb = sizeof ki;

  KbdGetStatus((KBDINFO far *) &ki, 0);

  showmask(ki.fsMask);
  show_shift_status(ki.fsState);
}

void showmask(unsigned mask)
{
  if(mask & 1) printf("echo enabled\n");
  if(mask & 2) printf("echo disabled\n");
  if(mask & 4) printf("mode is raw\n");
```

```
  if(mask & 8) printf("mode is cooked\n");
}

/* Display the shift status of the keyboard. */
void show_shift_status(unsigned status)
{
  printf("\n");
  if(status & 1) printf("Right shift pressed\n");
  if(status & 2) printf("Left shift pressed\n");
  if(status & 4) printf("A control key pressed\n");
  if(status & 8) printf("An Alt key pressed \n");
  if(status & 16) printf("Scroll Lock on\n");
  if(status & 32) printf("Num Lock on\n");
  if(status & 64) printf("Caps Lock on\n");
  if(status & 128) printf("Ins pressed\n");
  if(status & 256) printf("Left Control pressed\n");
  if(status & 512) printf("Left Alt key pressed\n");
  if(status & 1024) printf("Right Control key pressed\n");
  if(status & 2048) printf("Right Alt key pressed\n");
  if(status & 4096) printf("Scroll Lock pressed\n");
  if(status & 8192) printf("Num lock pressed\n");
  if(status & 16384) printf("Caps lock pressed\n");
  if(status & 32768) printf("SysRq key pressed\n\n");
}
```

You can set the status of the keyboard system using **KbdSetStatus**, whose prototype is

unsigned KbdGetStatus(KBDINFO far *info, unsigned handle);

where *info* points to a structure of type **KBDINFO**, which is the same as that defined for **KbdGetStatus**. In this service, *handle* must be 0.

Setting the status of most of the keyboard subsystem is a two-step process.

1. You set the proper bit in the **fsMask** variable of the *info* parameter. This tells OS/2 which type of function is going to be changed.
2. You set the value of the related parameter. For example, to turn Caps Lock on, first set bit 4 of **fsMask** to 1, then set bit 6 of **fsState**. When **KbdSetStatus** is called, the keyboard will be in Caps Lock mode.

The only exceptions to the two-step rule are switching between raw and cooked modes and switching between echo and no echo modes. For these you need only set the proper bits in the **fsMask** variable.

It is important to understand that a change in the status of the keyboard subsystem is local to the process that makes the change. For example, when the process terminates and the OS/2 command process resumes, the original default values are used.

The example that follows changes the turnaround character to a period and turns on Caps Lock mode. When this program ends, the carriage return automatically becomes the turnaround character again. Notice that the program first reads the status of the keyboard subsystem and then alters the value of the turnaround character before calling **KbdSetStatus**. The reason for this is that you need to preserve the state of the other KBD subsystem functions.

```
/* Using KbdSetStatus to change the turnaround character
   to a period and puts the keyboard into Caps Lock mode.
*/

#define INCL_SUB

#include <os2.h>


main()
{
  KBDINFO ki;
  char str[80];  /* input buffer */
  STRINGINBUF l;


  ki.cb = sizeof ki;

  /* change the turnaround char to a period. */
  KbdGetStatus((KBDINFO far *) &ki, 0);
  ki.chTurnAround = '.';
  /* Signal that a change to the turn around char is
     going to take place.
  */
  ki.fsMask = ki.fsMask | 64;

  ki.fsMask = ki.fsMask | 16; /* signal shift status change */
  ki.fsState = ki.fsState | 64; /* turn on Caps Lock */

  KbdSetStatus((KBDINFO far *) &ki, 0);

  /* demonstrate that new turnaround char is, indeed, active */
  printf("\nEnter a string; terminate with a period: ");
  l.cb = 80;
  KbdStringIn((char far *) str, (STRINGINBUF far *) &l,
              0, 0);
  str[l.cchIn] = '\0';  /* null terminate the string */
  printf("%d characters read, string is\n%s", l.cchIn, str);
}
```

READING A STRING USING KbdStringIn

Until now you have been reading only one character or scan code at a time from the keyboard. This is very useful, but OS/2 also provides a service that allows you to read a string of characters (without their associated scan codes). This service is called **KbdStringIn**, and its prototype is

```
unsigned KbdStringIn(char far *buf, STRINGINBUF far *len,
  unsigned wait, unsigned handle);
```

where *buf* is a pointer to the character array that will hold the string read from the keyboard. The *len* parameter is a structure of type **STRINGINBUF**, which takes this form:

```
typedef struct __STRINGINBUF {
        unsigned cb;     /* length of buffer */
        unsigned cchIn;  /* number of chars actually read */
} STRINGINBUF;
```

The **cb** field must hold the length of the array pointed to by *buf* prior to the call to **KbdStringIn**. The largest buffer you can use is 255 characters. Upon return **cchIn** holds the number of characters actually read from the keyboard.

The *wait* parameter determines what **KbdStringIn** does if no characters are present in the keyboard buffer. The effect is different in cooked and raw modes. In cooked mode (the default), the only allowed value of wait is 0, and **KbdStringIn** waits and reads characters until the user enters a carriage return. In raw mode, if *wait* is 0, **KbdStringIn** reads characters until the buffer pointed to by *buf* is completely full. If *wait* is 1, **KbdStringIn** reads however many characters are in the keyboard buffer (including zero characters) and returns immediately.

The following short program reads a string from the keyboard. It assumes that the default, cooked mode input is in use.

```
/* Reading a string from the keyboard using KbdStringIn. */

#define INCL_SUB

#include <os2.h>

main()
{
  char str[80];  /* input buffer */
  STRINGINBUF l;

  l.cb = 80;

  KbdStringIn((char far *) str, (STRINGINBUF far *) &l,
              0, 0);
  str[l.cchIn] = '\0';  /* null terminate the string */

  printf("%d characters read, string is\n%s", l.cchIn, str);
}
```

One advantage to using **KbdStringIn** in cooked mode is that you can use the standard editing keys to correct your entry before you press RETURN. Also, in cooked mode the characters you enter are automatically echoed to the screen.

5

USING THE MOUSE

There is no doubt that in the OS/2 environment the mouse will become a common, perhaps even an indispensible, accessory, partly because the Presentation Manager supports a graphics interface that lends itself to mouse operation. In the very near future, it will be the rare OS/2-compatible program that does not support mouse input.

There are really two complete sets of mouse interfacing services: those found in the core API and those defined by the Presentation Manager routines. You will use the Presentation Manager mouse services for most programming situations because they are designed to make menu selection and the like very easy. In a few types of applications, however, you may want to use the core mouse services. For example, if you are writing with a word processor that uses the entire screen and you simply want to provide mouse support for moving text around, the core API mouse services will require less overhead than the Presentation Manager equivalents. This chapter presents an overview of the core API mouse services. (The Presentation Manager is introduced in Part Three.)

All the core mouse services begin with the letters **Mou**. These services are listed and briefly described in Table 5-1.

Note: If you have programmed for the mouse in a DOS environment using Microsoft's MOUSE.LIB library, you may be surprised to learn that OS/2 uses a fundamentally different approach to mouse interfacing. In fact, except for the most general concepts, what you learned

Table 5-1. The Core Mouse Services

Service	Function
MouClose	Closes the mouse
MouDeRegister	Deactivates an alternative mouse service
MouDrawPtr	Displays the mouse pointer
MouFlushQue	Flushes the mouse information queue
MouGetDevStatus	Returns mouse status
MouGetEventMask	Returns mouse event mask
MouGetHotKey	Returns system hot key button
MouGetNumButtons	Returns the number of buttons on the mouse
MouGetNumMickeys	Returns the number of mickeys per centimeter
MouGetNumQueEl	Returns the number of information packets currently in the mouse queue
MouGetPtrPos	Returns the current location of the mouse pointer
MouGetPtrShape	Returns the shape of the mouse pointer
MouGetScaleFact	Returns the mouse movement scaling factors
MouInitReal	Initializes the real-mode mouse system
MouOpen	Opens the mouse
MouReadEventQue	Returns the next information packet in the mouse queue
MouRegister	Activates an alternative mouse function
MouRemovePtr	Removes the mouse pointer from the screen
MouSetDevStatus	Sets mouse device driver status information
MouSetEventMask	Sets the mouse event mask
MouSetHotKey	Sets the system hot key
MouSetPtrPos	Sets the mouse pointer's screen position
MouSetPtrShape	Sets the shape of the mouse pointer
MouSetScaleFact	Sets the mouse movement scale factor
MouSynch	Synchronizes mouse access

about interfacing to the mouse under DOS has little applicability to the OS/2 mouse interface.

THE MOUSE

Before your programs can use the mouse services, OS/2 must have loaded two device drivers called MOUSEB05.SYS and POINTDD.SYS. The OS/2 setup program automatically makes the proper entries in your CONFIG.SYS file that cause these device drivers to be loaded.

However, if these device drivers are not specified in the CONFIG.SYS file, add the following lines to your CONFIG.SYS file:

```
DEVICE=C:\OS2\POINTDD.SYS
DEVICE=C:\OS2\MOUSEB05.SYS
```

MOUSE BASICS

Unlike the screen and keyboard services, the mouse services cannot use the default handle 0. The first thing your program must do to support the mouse is to open the mouse by a means of a call to **MouOpen**, which returns a valid mouse handle. You must then use this handle with all other mouse services.

For various reasons, most of which have to do with the fact that OS/2 is a multitasking system, information about the mouse is kept in a queue until it is read by your program. The mouse queues are first-in, first-out. Each time you press a button or move the mouse, a hardware interrupt transfers control to the mouse device driver. The device driver determines what has happened and generates an information packet that is put in the queue. The packet includes such things as the current position of the mouse and which buttons are pressed. The queue is not very long, so if a large number of packets are generated before your program reads them, some of the packets may be overwritten. A queue overrun generally causes no real harm.

Although the mouse and the screen are fundamentally separate devices, the OS/2 core mouse services can provide the appearance of a strong link between the two. For example, the mouse pointer is automatically moved about the screen when you move the mouse. (The mouse pointer is the symbol on the screen that shows the mouse's current screen position.) In essence, you think of the mouse as being on the screen rather than on the desk. In the default mode of operation, the mouse services also return the row and column position of the mouse pointer.

In text mode, the mouse pointer is a solid block. It is possible to change the shape of the pointer if the screen is in a graphics mode. However, since OS/2 most easily supports graphics through the Presentation Manager, you will probably never use the core mouse services in a graphics mode.

The mouse services can return position information about the mouse in one of two ways:

1. In the default mode of operation the mouse services return the row and column coordinates of the pointer. The pointer is always moved in screen units. For text modes, this means a character position. In a graphics mode it means a pel. There is no concept, for example, of the pointer being "between" two screen units. All coordinates are relative to the upper left corner of the screen, which is 0,0. All the examples in this chapter use this mode because it is by far the easiest to work with for text mode applications.
2. The services can also return position information in mickey counts. The *mickey* is the basic unit of mouse movement and commonly equals approximately 1/120 inch. Two mickey counts are returned: one for the x and one for the y coordinate. If the y-coordinate mickey count is negative, the mouse has moved forward (away from you) on the desk and the pointer has moved up the screen that number of mickeys. A positive y-coordinate value indicates that the mouse has moved toward you and the pointer has moved down the screen. A negative x-coordinate value means that the mouse and pointer have moved to the left; a positive value means that they have moved to the right.

The standard IBM/Microsoft mouse has two buttons. However, it is possible to have mice connected to the system that have either one or three buttons instead. The mouse subsystem can operate with one-, two-, or three-button mice. However, your program may have to make explicit provisions for such possibilities. The leftmost button is always button number one.

OPENING THE MOUSE

Before the mouse can be used, it must be opened using **MouOpen**. The prototype for **MouOpen** is

```
signed MouOpen(char far *driver, unsigned short far *mhandle)
```

where *driver* is a pointer to a null-terminated string that contains the name of the mouse pointer device driver. You can cause the mouse

system to use its default pointer driver by passing a null in this parameter. This is useful when the name of the device driver is not known. The default driver will be used in all examples in this chapter. On return the variable pointed to by *mhandle* contains the current mouse handle.

The call to **MouOpen** essentially intializes the mouse system for use. It does not display the mouse pointer or return any status information about the mouse.

DISPLAYING THE MOUSE POINTER

Once the mouse is opened for use, one of the first things you will probably want to do is have the pointer displayed on the screen. To do this you use the **MouDrawPtr** service, which has the prototype

```
unsigned MouDrawPtr(unsigned short mhandle);
```

where *mhandle* is a valid handle returned by **MouOpen**. In text modes the pointer is a solid block.

POSITIONING THE MOUSE POINTER

The mouse system automatically moves the pointer around on the screen when you move the mouse on your desk. Your program does not need to move the pointer explicitly unless you want it to. For several reasons you may wish to reposition the mouse pointer on the screen. For example, you may need to move the pointer to the top of a pop-up menu. When you explicitly move the mouse pointer, the mouse subsystem automatically updates all of its location and status information so that the next time you move the mouse, the pointer is moved relative to its new screen position. The core mouse service that positions the mouse pointer is called **MouSetPtrPos**, and its prototype is

```
unsigned MouSetPtrPos( (PTRLOC far *) loc,
        unsigned short mhandle);
```

Pointer *loc* points to a structure of type **PTRLOC**, which is defined as

```
typedef strict _PTRLOC {
        unsigned row;
        unsigned col;
} PTRLOC;
```

The values of **row** and **col** must be within the range defined by the current video mode. For the default 80 by 25 text mode, the range for **row** is 0 through 24; the range for **col** is 0 through 79.

CREATING A MOUSE INITIALIZATION FUNCTION

Before going further with our discussion of the mouse, let's create a mouse initialization function that opens the mouse, positions the mouse pointer at the upper left corner (location 0,0), and draws the pointer. The function also returns the mouse handle to the calling routine. The function is called **initmouse()** and is shown here:

```
/* Open the mouse, draw the pointer, and position
   the mouse at the upper left corner.
*/
unsigned short initmouse()
{
  unsigned short mhandle;
  unsigned short err;
  PTRLOC p;

  err = MouOpen((char far *) 0, (unsigned short far *) &mhandle);
  if(err) {
    printf("%d error in opening mouse\n", err);
    return 0;
  }

  /* position the mouse pointer in the upper left corner */
  p.row = 0;
  p.col = 0;
  MouSetPtrPos((PTRLOC far *) &p, mhandle);

  /* make the pointer visible */
  MouDrawPtr(mhandle);

  return mhandle;
}
```

The **initmouse()** function is used in all the example programs in this chapter to facilitate the initialization of the mouse driver. For your own applications you may need to change the initial location of the mouse pointer.

SENSING MOUSE MOVEMENT AND BUTTON PRESSES

Each time you move the mouse or press a button, an event information packet is put on the end of the mouse queue. Your program reads information from the queue using the **MouReadEventQue** function, which has the prototype

```
unsigned MouReadEventQue(MOUEVENTINFO far *status,
        unsigned far *wait, unsigned short mhandle);
```

where *status* points to an event structure of type **MOUEVENTINFO**, which is defined as

```
typedef struct _MOUEVENTINFO {
        unsigned fs; /* encoded state of the mouse */
        unsigned long Time; /* time when event occurred */
        unsigned row; /* row position of mouse pointer */
        unsigned col; /* col position of mouse pointer */
} MOUEVENTINFO;
```

The structure pointed to by *status* holds event information about the mouse when **MouReadEventQue** returns.

On return from the call, the **fs** field is encoded like this*:

Bit	Meaning When Set
0	Mouse moved, no buttons pressed
1	Mouse moved, button 1 pressed
2	Button 1 pressed, no movement
3	Mouse moved, button 2 pressed
4	Button 2 pressed, no movement
5	Mouse moved, button 3 pressed
6	Button 3 pressed, no movement
7	Reserved, always 0

As you can see, by examining this field it is possible to detect whether the mouse has moved and if a button is depressed. (Remember that the standard IBM/Microsoft mouse has two buttons, but other types of

*The following table is adapted from tables in *Operating System/2 Programmer's Reference Manual*, with permission of Microsoft Corporation.

mice can be connected to the system. Later you will learn how to detect the number of buttons a given mouse has.) If **fs** is 0, no event has occurred.

The **Time** field represents the system time at which the event occurred; the value is in milliseconds. The **row** and **col** fields hold the screen location of the mouse pointer. By default these values are in screen units, but you can set the mouse subsystem to report the value in mickeys.

The value of the *wait* parameter to **MouReadEventQue** determines whether the service waits for an event packet if one is not waiting in the queue. If *wait* is 0, the function returns immediately if nothing is in the queue, filling the information structure with zeros. If *wait* is 1, the service waits until a mouse event is generated.

The **mhandle** parameter must be a valid mouse handle.

You can use the **MouReadEventQue** service to write a short program that displays the mouse event packet. First you need a routine that decodes the event information and displays it on the screen. The function **show_mouse_state()** shown here accomplishes this:

```
/* Show the current location of the mouse and which
   buttons are pressed.
*/
void show_mouse_state(MOUEVENTINFO state)
{
  char cell[2];

  VioSetCurPos(10, 0, 0);

  /* clear a small part of the screen */
  cell[0] = ' '; cell[1] = 7;
  VioScrollDn(10, 0, 12, 79, 3, (char far *) cell, 0);

  /* decode button press information */
  if((state.fs & 2) || (state.fs & 4))
    printf("button one is down\n");
  if((state.fs & 8) || (state.fs & 16))
    printf("button two is down\n");
  if((state.fs & 32) || (state.fs & 64))
    printf("button three is down\n");

  /* see if the mouse has moved */
  if((state.fs & 1) || (state.fs & 2) || (state.fs & 8) ||
     (state.fs & 32))
       printf("the mouse has moved");

  /* display current position and time of event */
  VioSetCurPos(15, 0, 0);
  printf("                          ");
  VioSetCurPos(15, 0, 0);
  printf("%d %d time: %ld", state.row, state.col, state.Time);
}
```

Using the **show_mouse_state()** function, the program that follows displays the information packet generated by each mouse event. Notice that it waits for a packet if one is not already waiting in the queue.

```
/* Demonstrate how to access the mouse and decode the
   status information returned by it.
*/

#define INCL_SUB

#include <os2.h>

unsigned short initmouse(void);
void show_mouse_state(MOUEVENTINFO);
void clrscr(void);

main()
{
  unsigned short mhandle;
  unsigned state;
  unsigned wait;
  MOUEVENTINFO info;

  clrscr();

  mhandle = initmouse();

  do {
    wait = 1;
    MouReadEventQue((MOUEVENTINFO far *) &info,
                    (unsigned far *) &wait, mhandle);
    show_mouse_state(info);

  } while (!kbhit());
  MouClose(mhandle);

}

/* Open the mouse, draw the pointer, and position
   the mouse at the upper left corner.
*/
unsigned short initmouse()
{
  unsigned short mhandle;
  unsigned short err;
  PTRLOC p;

  err = MouOpen((char far *) 0, (unsigned short far *) &mhandle);
  if(err) {
    printf("%d error in opening mouse\n", err);
    return 0;
  }
  p.row = 0;
  p.col = 0;
  MouSetPtrPos((PTRLOC far *) &p, mhandle);
  MouDrawPtr(mhandle);
  return mhandle;
}
```

```
/* Show the current location of the mouse and which
   buttons are pressed.
*/
void show_mouse_state(MOUEVENTINFO state)
{
  char cell[2];

  VioSetCurPos(10, 0, 0);

  /* clear a small part of the screen */
  cell[0] = ' '; cell[1] = 7;
  VioScrollDn(10, 0, 12, 79, 3, (char far *) cell, 0);

  /* decode button press information */
  if((state.fs & 2) || (state.fs & 4))
    printf("button one is down\n");
  if((state.fs & 8) || (state.fs & 16))
    printf("button two is down\n");
  if((state.fs & 32) || (state.fs & 64))
    printf("button three is down\n");

  /* see if the mouse has moved */
  if((state.fs & 1) || (state.fs & 2) || (state.fs & 8) ||
     (state.fs & 32))
       printf("the mouse has moved");

  /* display current position and time of event */
  VioSetCurPos(15, 0, 0);
  printf("                              ");
  VioSetCurPos(15, 0, 0);
  printf("%d %d time: %ld", state.row, state.col, state.Time);
}
/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

To stop the program, press any key on the keyboard. The program ends when the next mouse event occurs.

SOME CUSTOM FUNCTIONS TO INTERROGATE THE MOUSE

You can create some simple functions to facilitate checking for mouse movement or button presses. These functions are shown here. (They assume that a standard two-button mouse is installed in the system.

You can easily change this as required by your system.)

```
/* Return true if left button is pressed. */
leftbutton(MOUEVENTINFO info)
{
  return((info.fs & 2) || (info.fs & 4));
}

/* Return true if right button is pressed. */
rightbutton(MOUEVENTINFO info)
{
  return((info.fs & 8) || (info.fs & 16));
}


/* Return true if mouse has moved. */
mousemoved(MOUEVENTINFO info)
{
  return((info.fs & 1) || (info.fs & 2) || (info.fs & 8) ||
         (info.fs & 32));
}
```

The functions are passed an information packet returned by **MouReadEventQue** elsewhere in any program that uses them.

This program illustrates how to make use of the custom functions to show when the mouse is moved or a button is pressed.

```
/* This program illustrates how you can create custom mouse
   functions which can make your application programs easier
   to write.
*/

#define INCL_SUB

#include <os2.h>

unsigned short initmouse(void);
int leftbutton(MOUEVENTINFO);
int rightbutton(MOUEVENTINFO);
int mousemoved(MOUEVENTINFO);
void clrscr(void);

main()
{
  unsigned short mhandle;
  unsigned state;
  MOUEVENTINFO info;
  unsigned wait;

  clrscr();
  /* open the mouse, show the pointer, and position
     the mouse at the upper left corner
  */
  mhandle = initmouse();
  if(!mhandle) exit(1); /* error opening mouse */
```

```
    /* monitor the mouse and report any activity */
    do {
      wait = 0;
      MouReadEventQue((MOUEVENTINFO far *) &info,
                      (unsigned far *) &wait, mhandle);
      if(leftbutton(info)) printf("left button\n");
      if(rightbutton(info)) printf("right button\n");
      if(mousemoved(info)) printf("mouse moved\n");
    } while (!kbhit());
    MouClose(mhandle);
}

/* Open the mouse, draw the pointer, and position
   the mouse at the upper left corner.
*/
unsigned short initmouse()
{
    unsigned short mhandle;
    unsigned short err;
    PTRLOC p;

    err = MouOpen((char far *) 0, (unsigned short far *) &mhandle);
    if(err) {
      printf("error in opening mouse\n");
      return 0;
    }
    p.row = 0;
    p.col = 0;
    MouSetPtrPos((PTRLOC far *) &p, mhandle);
    MouDrawPtr(mhandle);
    return mhandle;
}

/* Return true if left button is pressed. */
leftbutton(MOUEVENTINFO info)
{
    return((info.fs & 2) || (info.fs & 4));
}

/* Return true if right button is pressed. */
rightbutton(MOUEVENTINFO info)
{
    return((info.fs & 8) || (info.fs & 16));
}

/* Return true if mouse has moved. */
mousemoved(MOUEVENTINFO info)
{
    return((info.fs & 1) || (info.fs & 2) || (info.fs & 8) ||
           (info.fs & 32));
}


/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
    char space[2];

    space[0] = ' ';
```

```
    space[1] = 7;
    VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

CHANGING THE SCALING FACTORS

Each time you move the mouse, an absolute amount of distance is displaced. However, how the physical distance you move the mouse on the desk is transformed into movement of the pointer is controlled by the value of the row and column coordinate scaling factors. The scaling factors determine how many mickeys the mouse must be moved in order to change the screen location of the mouse pointer by one unit. That is, a scaling factor of 1 means that for each mickey the mouse is moved, the mouse pointer moves one screen unit. If the scaling factor is 2, the mouse pointer is moved one screen unit for every 2 mickeys that the mouse is moved. The greater the scaling factor, the more the mouse has to be physically moved on the desk to move the pointer to the next screen unit. What values of scaling units make the best conversion ratio is subject to intense debate. To some extent, the choice of a scaling factor is governed more by the amount of free desk space than by preference! The larger the scaling factor, the more space is needed.

You can determine the current scaling factors by using the **MouGetScaleFact** service, which has the prototype

```
unsigned MouGetScaleFact(SCALEFACT far *fact,
        unsigned short mhandle);
```

The *fact* parameter is a pointer to a structure of type **SCALEFACT**, which is defined

```
typedef struct _SCALEFACT {
        unsigned rowScale; /* row scaling factor */
        unsigned colScale; /* column scaling factor */
} SCALEFACT;
```

The fields **rowScale** and **colScale** hold the current row and column scaling factors of the mouse subsystem.

The mhandle parameter is the handle returned by **MouOpen**.

To set the scaling factors, use **MouSetScaleFact**, whose prototype is

```
unsigned MouSetScaleFact(SCALEFACT far *fact,
        unsigned short mhandle);
```

The structure pointed to by *fact* is as previously defined. The valid scaling factors for both row and column directions are 1 through 32,767. However, a practical range is loosely 1 through 24.

Although this is not directly related to scaling factors, it is sometimes interesting to know how many centimeters the mouse has moved. To determine this, you must first call **MouGetNumMickeys**) which returns the number of mickeys in a centimeter. Although generally a mickey is about 1/120 inch, it is not an absolute value. If your application must know how far the mouse has actually moved, you must call **MouGetNumMickeys** to know for sure. The prototype for **MouGetNumMickeys** is

```
unsigned MouGetNumMickeys(unsigned far *mickeys,
        unsigned short mhandle);
```

On return the integer pointed to by *mickeys* holds the number of mickeys per centimeter.

The demonstration program shown here displays the number of mickeys per centimeter followed by the default row and column scaling factors. Next the scaling factors are set to their lowest value: 1. After that each time you press the left button, the row factor increases; each time you press the right button, the column factor increases. In this way, you can experiment with different scaling factors to see which combination provides the most pleasing effects.

```
/* Demonstrate scaling factors.
*/

#define INCL_SUB

#include <os2.h>

unsigned short initmouse(void);
unsigned getmickeys(unsigned short);
int leftbutton(unsigned short, MOUEVENTINFO);
int rightbutton(unsigned short, MOUEVENTINFO);
void clrscr(void);

main()
{
```

```
    unsigned short mhandle;
    unsigned state;
    unsigned wait;
    MOUEVENTINFO info;
    SCALEFACT sf;
    unsigned rscale, cscale;
    char changed;

    clrscr();

    mhandle = initmouse();
    if(!mhandle) exit(1); /* error opening mouse */

    VioSetCurPos(0, 0, 0);

    /* show mickeys per centimeter */
    printf("mickeys per centimeter: %d\n", getmickeys(mhandle));
    MouGetScaleFact((SCALEFACT far *) &sf, mhandle);
    printf("default row factor: %d   column factor: %d\n",
            sf.rowScale, sf.colScale);

    rscale = cscale = 1; /* start scaling at lowest ratio */
    sf.rowScale = rscale; sf.colScale = cscale;
    MouSetScaleFact((SCALEFACT far *) &sf, mhandle);

    do {
      changed = 0;
      wait = 1;
      MouReadEventQue((MOUEVENTINFO far *) &info,
                      (unsigned far *) &wait, mhandle);

      /* Press left button to increase the row scale factor.
         Press right button to increase the column scale factor.
      */
      if(leftbutton(mhandle, info)) {
        rscale++;
        changed = 1;
      }
      if(rightbutton(mhandle, info)) {
        cscale++;
        changed = 1;
      }

      if(changed) {
        sf.rowScale = rscale; sf.colScale = cscale;
        MouSetScaleFact((SCALEFACT far *) &sf, mhandle);
        VioSetCurPos(2, 0, 0);
        printf("row scale: %d  column scale: %d", rscale, cscale);
      }
    } while (!kbhit());
    MouClose(mhandle);
}

/* Open the mouse, draw the pointer, and position
   the mouse at the upper left corner.
*/
unsigned short initmouse()
{
    unsigned short mhandle;
    unsigned short err;
    PTRLOC p;
```

```
    err = MouOpen((char far *) 0, (unsigned short far *) &mhandle);
    if(err) {
      printf("error in opening mouse\n");
      return 0;
    }
    p.row = 0;
    p.col = 0;
    MouSetPtrPos((PTRLOC far *) &p, mhandle);
    MouDrawPtr(mhandle);
    return mhandle;
}

/* Return the number of mickeys per centimeter. */
unsigned getmickeys(unsigned short mhandle)
{
    unsigned mick;

    MouGetNumMickeys((unsigned far *) &mick, mhandle);
    return mick;
}

/* Return true if left button is pressed. */
leftbutton(unsigned short mhandle, MOUEVENTINFO info)
{
    return((info.fs & 2) || (info.fs & 4));
}

/* Return true if right button is pressed. */
rightbutton(unsigned short mhandle,  MOUEVENTINFO info)
{
    return((info.fs & 8) || (info.fs & 16));
}

/* A simple way to clear the screen by filling
    it with spaces.
*/
void clrscr()
{
    char space[2];

    space[0] = ' ';
    space[1] = 7;
    VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

Measuring Distance with the Mouse

Because you can know the number of mickeys per centimeter, you can use the mouse to measure distance, on a map for example, by multiplying the number of mickeys per centimeter by the scaling factor and by the number of screen units the mouse pointer moves. Expressed in mathematical notation, the formula is

$$\text{distance} = (\text{mickeys/centimeter}) * \text{scale factor} * \text{screen units}$$

When the scaling factor is 1, the formula is simply mickeys per centimeter times the number of screen units. The following program uses this formula to compute the number of centimeters the mouse has moved in a vertical direction. To use the program, locate the mouse at the start of the distance you wish to measure and press the left button. Next, move the mouse to the end of the distance and press the right button. The number of centimeters covered by the mouse will be displayed.

```
/* This program uses the mouse to measure distance. */

#define INCL_SUB

#include <os2.h>

unsigned short initmouse(void);
unsigned getmickeys(unsigned short);
int leftbutton(unsigned short, MOUEVENTINFO);
int rightbutton(unsigned short, MOUEVENTINFO);
void clrscr(void);

main()
{
  unsigned short mhandle;
  unsigned state;
  unsigned wait;
  MOUEVENTINFO info;
  SCALEFACT sf;
  unsigned startrow, endrow, rscale, cscale;

  clrscr();
  /* open the mouse, show the pointer, and position
     the mouse at the upper left corner
  */

  printf("press left button to start measuring\n");
  printf("press right button to stop measuring");

  mhandle = initmouse();
  if(!mhandle) exit(1); /* error opening mouse */

  VioSetCurPos(0, 0, 0);

  rscale = cscale = 1; /* start scaling at lowest ratio */
  sf.rowScale = rscale; sf.colScale = cscale;
  MouSetScaleFact((SCALEFACT far *) &sf, mhandle);

  wait = 1;
  do {
    MouReadEventQue((MOUEVENTINFO far *) &info,
                    (unsigned far *) &wait, mhandle);

    /* start reading distance */
    if(leftbutton(mhandle, info)) {
      startrow = info.row;
    }
```

```
    /* stop reading distance */
    if(rightbutton(mhandle, info)) {
      endrow = info.row;
      printf("%d centimeters moved\n",
             (endrow-startrow)*getmickeys(mhandle));
    }

  } while (!kbhit());
  MouClose(mhandle);
}

/* Open the mouse, draw the pointer, and position
   the mouse at the upper left corner.
*/
unsigned short initmouse()
{
  unsigned short mhandle;
  unsigned short err;
  PTRLOC p;

  err = MouOpen((char far *) 0, (unsigned short far *) &mhandle);
  if(err) {
    printf("error in opening mouse\n");
    return 0;
  }
  p.row = 0;
  p.col = 0;
  MouSetPtrPos((PTRLOC far *) &p, mhandle);
  MouDrawPtr(mhandle);
  return mhandle;
}

/* Return the number of mickeys per centimeter. */
unsigned getmickeys(unsigned short mhandle)
{
  unsigned mick;

  MouGetNumMickeys((unsigned far *) &mick, mhandle);
  return mick;
}

/* Return true if left button is pressed. */
leftbutton(unsigned short mhandle, MOUEVENTINFO info)
{
  return((info.fs & 2) || (info.fs & 4));
}

/* Return true if right button is pressed. */
rightbutton(unsigned short mhandle,  MOUEVENTINFO info)
{
  return((info.fs & 8) || (info.fs & 16));
}

/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
  char space[2];
```

```
  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

DETERMINING THE NUMBER OF BUTTONS

Your program can find out how many buttons are on the mouse connected to the system by using the **MouGetNumButtons** service, which has the prototype

```
unsigned MouGetNumButtons(unsigned far *b,
        unsigned short mhandle);
```

On return from the call, the integer pointed to by *b* contains a value equal to the number of buttons on the mouse. The following program demonstrates this service's use:

```
/* Display the number of buttons on the mouse. */

#define INCL_SUB

#include <os2.h>

unsigned short initmouse(void);

main()
{
  unsigned short mhandle;
  unsigned button;

  mhandle = initmouse();
  if(!mhandle) exit(1); /* error opening mouse */

  MouGetNumButtons((unsigned far *) &button, mhandle);
  printf("Your mouse has %d buttons.\n", button);
  MouClose(mhandle);
}

/* Open the mouse, draw the pointer, and position
   the mouse at the upper left corner.
*/
unsigned short initmouse()
{
```

```
  unsigned short mhandle;
  unsigned short err;
  PTRLOC p;

  err = MouOpen((char far *) 0, (unsigned short far *) &mhandle);
  if(err) {
    printf("error in opening mouse\n");
    return 0;
  }
  p.row = 0;
  p.col = 0;
  MouSetPtrPos((PTRLOC far *) &p, mhandle);
  MouDrawPtr(mhandle);
  return mhandle;
}
```

FLUSHING THE QUEUE

You may need to clear the contents of the queue. For example, if the user requests that the mouse be used for a new purpose, any current contents of the queue need to be cleared. To accomplish this task OS/2 provides the **MouFlushQue** service, which has the prototype

```
unsigned MouFlushQue(unsigned short mhandle);
```

Although none of the examples in this chapter use this service, it is available if your programs need it.

A SIMPLE MOUSE MENU EXAMPLE

The mouse is commonly used as an input device for menu selection. Although you will generally use the Presentation Manager mouse services when working with menus, the following example illustrates how you can accomplish menu selection by using only the core mouse services.

The key to using the mouse for menu selection is to convert the mouse's position into an integer that represents a menu item. For example, if a menu has three selections, the first could be identified with the number 0, the second with 1, and the third with 2. The trick, of course, is to transform the mouse's position into one of these numbers. One easy way to do this is to display all menu entries vertically (in a list) and then simply use the current row position of the

mouse pointer (less an appropriate offset from 0) to identify the menu selection. For example, if the menu begins on line 10 and the mouse is on line 11 when the selection is made, the second menu item is chosen because 11 − 10 is 1. Here 10 is the offset used to normalize the row position. The offset is always the row number of the first entry in the menu.

The way menu selections are generally made with the mouse is by pressing a button. In the example developed here, the left button is used.

The function **get_menu_select()** is passed the upper left coordinate of the first entry in the menu, the number of items in the menu, the width in characters of the longest entry in the menu, and the activate mouse handle. It then positions the mouse at the top of the menu and waits for a selection to be made. Notice that it does not allow the mouse to leave the area defined by the menu until a selection is made. This sort of restriction is not mandatory, but it is very common because it simplifies your application program.

```
/* This function positions the mouse pointer at the top
   of the specified area and keeps the mouse confined to
   those rows that have menu entries.  The parameters x and y
   specify the upper left corner of the menu, the len parameter
   specifies the number of menu entries, and the width
   parameter specifies the width of the longest menu entry.
*/
get_menu_select(unsigned x, unsigned y, int len, int width,
                unsigned short mhandle)
{
  MOUEVENTINFO info;
  unsigned wait;
  PTRLOC p;

  p.row = y;  p.col = x;

  MouSetPtrPos((PTRLOC far *) &p, mhandle);
  for(;;) {
    wait = 1;
    MouReadEventQue((MOUEVENTINFO far *) &info,
                    (unsigned far *) &wait, mhandle);
    if(info.row<y) MouSetPtrPos((PTRLOC far *) &p, mhandle);
    if(info.row>=y+len) MouSetPtrPos((PTRLOC far *) &p, mhandle);
    if(info.col<x) MouSetPtrPos((PTRLOC far *) &p, mhandle);
    if(info.col>=x+width) MouSetPtrPos((PTRLOC far *) &p, mhandle);
    if((info.fs & 2) || (info.fs & 4))
       return info.row-y;
  }

}
```

The following program demonstrates how the **get_menu_select()** function can be used:

```
/* This program illustrates how to use the mouse to make a
   menu selection.
*/

#define INCL_SUB

#include <os2.h>

unsigned short initmouse(void);
void clrscr(void), display_menu(void);
int get_menu_select(unsigned, unsigned, int, int, unsigned short);

main()
{
  unsigned short mhandle;
  unsigned state;
  unsigned wait;
  MOUEVENTINFO info;

  clrscr();

  mhandle = initmouse();
  if(!mhandle) exit(1); /* error opening mouse */

  display_menu();

  printf("You chose item number %d",
    get_menu_select(0, 5, 3, 7, mhandle));

  MouClose(mhandle);
}

/* Open the mouse, draw the pointer, and position
   the mouse at the upper left corner.
*/
unsigned short initmouse()
{
  unsigned short mhandle;
  unsigned short err;
  PTRLOC p;

  err = MouOpen((char far *) 0, (unsigned short far *) &mhandle);
  if(err) {
    printf("error in opening mouse\n");
    return 0;
  }
  p.row = 0;
  p.col = 0;
  MouSetPtrPos((PTRLOC far *) &p, mhandle);
  MouDrawPtr(mhandle);
  return mhandle;
}
```

```
/* This function positions the mouse pointer at the top
   of the specified area and keeps the mouse confined to
   those rows that have menu entries.  The parameters x and y
   specify the upper left corner of the menu and the len parameter
   specifies the number of menu entries.
*/
get_menu_select(unsigned x, unsigned y, int len, int width,
                unsigned short mhandle)
{
  MOUEVENTINFO info;
  unsigned wait;
  PTRLOC p;


  p.row = y;  p.col = x;

  MouSetPtrPos((PTRLOC far *) &p, mhandle);
  for(;;) {
    wait = 1;
    MouReadEventQue((MOUEVENTINFO far *) &info,
                    (unsigned far *) &wait, mhandle);
    if(info.row<y) MouSetPtrPos((PTRLOC far *) &p, mhandle);
    if(info.row>=y+len) MouSetPtrPos((PTRLOC far *) &p, mhandle);
    if(info.col<x) MouSetPtrPos((PTRLOC far *) &p, mhandle);
    if(info.col>=x+width) MouSetPtrPos((PTRLOC far *) &p, mhandle);
    if((info.fs & 2) || (info.fs & 4))
       return info.row-y;
  }

}


/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}


/* Display a menu. */
void display_menu()
{
  VioSetCurPos(5, 0, 0);
  printf("Apples\n");
  printf("Oranges\n");
  printf("Grapes\n");
  printf("\nMake a selection\n");
}
```

The approach to the menu and the menu selection process used in this example is simple but effective. If you are interested in such things

as pop-up and pull-down menus, however, you will want to consult *C: Power User's Guide* by Herbert Schildt (Osborne/McGraw-Hill, 1987), which covers this subject and several other interesting and difficult programming issues.

A VARIATION ON THE PING-PONG VIDEO GAME

You probably remember the very first video games. Very crude by today's standards, they were essentially games of ping-pong. This chapter on the core mouse services ends with a variation of the old ping-pong game that uses the mouse to control the "paddle" (the mouse's pointer). The game works like this. A ball, represented by an asterisk, bounces around the screen moving left to right. When the ball hits the top or the bottom of the screen it reverses its vertical direction. If the paddle hits the ball, the ball also reverses its vertical direction. The computer scores a point each time the ball hits the bottom of the screen. You score a point each time the ball hits the center of the top of the screen. Only character positions 40 through 60 score points for you. (This limitation is added to balance the game between you and the computer.) The positions at the top of the screen that do not score points are shown by a dashed line. The unmarked area is the goal. The computer's score is shown in the lower left corner; yours is shown in the lower right. The game runs continuously until you press a key on the keyboard.

Although the program is fairly straightforward, a few key points are worth mentioning. First, the cursor (not the mouse pointer) is hidden so that it won't detract from the playing action. The ball is moved only once each 20 times the main loop executes. The **toggle** variable is used to control this value, which slows the ball down enough for a human to "hit" it. Depending on your computer's speed, you may need to change this value. To give the appearance of movement, the ball is erased from its current position before being moved to the next screen unit. Finally, the values of **deltax** and **deltay** control the angle of the ball as it moves about the screen.

The program is presented here for your amusement, without further comment:

```
/* A simple version of the old Ping-Pong video game.
*/

#define INCL_SUB
#define INCL_DOS

#include <os2.h>

unsigned short initmouse(void);
void movemouse(MOUEVENTINFO, unsigned short);
void clrscr(void), moveball(void), display_score(void);

unsigned row = 0, col = 0;
unsigned ballx, bally;
int deltax, deltay;
int computer=0, user=0, oldcomputer=-1; olduser=-1;
main()
{
  unsigned short mhandle;
  unsigned state;
  unsigned wait;
  MOUEVENTINFO info;
  SCALEFACT sf;
  int theta;
  VIOCURSORINFO c;
  unsigned toggle;
  PTRLOC p;

  clrscr();

  mhandle = initmouse();
  if(!mhandle) exit(1); /* error opening mouse */

  sf.rowScale = sf.colScale = 2;
  MouSetScaleFact((SCALEFACT far *) &sf, mhandle);

  /* hide the cursor */
  VioGetCurType((VIOCURSORINFO far *) &c, 0);
  c.attr = -1;
  VioSetCurType((VIOCURSORINFO far *) &c, 0);

  ballx = 10; bally = 1;
  deltax = 1; deltay = 1;
  toggle = 0;
  theta = 2;

  /* Draw goal line */
  VioSetCurPos(0, 0, 0);
  printf("------------------------------");
  VioSetCurPos(0, 50, 0);
  printf("------------------------------");
  wait = 0;
  VioSetCurPos(bally, ballx, 0); printf("*");

  do {
    MouReadEventQue((MOUEVENTINFO far *) &info,
                    (unsigned far *) &wait, mhandle);
```

```
    /* If there has been a change in the mouse's position,
       update the counters.
    */
    if(info.fs) {
      row = info.row;
      col = info.col;
    }
    /* If mouse pointer intersects the ball, reverse vertical
       direction.
    */
    if(ballx==col && bally==row) {
      p.row = row; p.col = col+1;
      MouSetPtrPos((PTRLOC far *) &p, mhandle);
      deltay = -deltay;
      moveball();
      DosBeep(500, 50);
    }

    if(!(toggle%20)) moveball();
    toggle++;

    display_score();
  } while (!kbhit());
  MouClose(mhandle);

  /* restore the cursor */
  c.attr = 0;
  VioSetCurType((VIOCURSORINFO far *) &c, 0);
}

/* Open the mouse, draw the pointer, and position
   the mouse at the upper left corner.
*/
unsigned short initmouse()
{
  unsigned short mhandle;
  unsigned short err;
  PTRLOC p;

  err = MouOpen((char far *) 0, (unsigned short far *) &mhandle);
  if(err) {
    printf("error in opening mouse\n");
    return 0;
  }
  p.row = 1;
  p.col = 0;
  MouSetPtrPos((PTRLOC far *) &p, mhandle);
  MouDrawPtr(mhandle);
  return mhandle;
}

/* Move the Ball. */
void moveball()
{
  static int toggle2=0;
  int i;

  VioSetCurPos(bally, ballx, 0); printf(" ");
  if(toggle2) ballx += deltax;
```

```
  bally += deltay;
  if(ballx == 80) ballx = 0;
  if(bally >=  24) {
    if(deltay>0) deltay = -deltay;
    computer++; /* give a point to the computer */
    DosBeep(300, 50);
  }
  if(bally == 1) {
    if(deltay<0) deltay = -deltay;
    if(ballx >30 && ballx <50) {
      user++; /* give a point to the user */
      for(i=0; i<5; i++ ) DosBeep(300+(i*100), 50);
    }
    else DosBeep(300, 50);
  }
  toggle2 = !toggle2;
  VioSetCurPos(bally, ballx, 0); printf("*");
}

/* Display the score. */
void display_score()
{
  /* don't waste time redisplaying unchanged score */
  if(computer==oldcomputer && user==olduser) return;

  VioSetCurPos(24, 0, 0);
  printf("  ");
  VioSetCurPos(24, 0, 0);
  printf("%d", computer);
  VioSetCurPos(24, 76, 0);
  printf("   ");
  VioSetCurPos(24, 76, 0);
  printf("%d", user);
  oldcomputer = computer;
  olduser = user;
}

/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

6

FILE I/O

The OS/2 file I/O subsystem is an amazingly straightforward and efficient way to access disk files and other devices. At its core are four services: **DosOpen**, **DosRead**, **DosWrite**, and **DosClose**. If you are familiar with C's unbuffered I/O system, you will be pleased to learn that these services parallel **open()**, **read()**, **write()**, and **close()**. In fact, many of the file services are similar to C's I/O functions. Even if you are unfamiliar with these C functions, the OS/2 file system is very easy to learn and use.

The OS/2 file I/O services are shown and briefly described in Table 6-1. Notice that all the functions begin with the prefix **Dos**.

As has been the case with many of the OS/2 services, the OS/2 file system is closely paralleled by the C file system. For most low-performance applications you will probably use the C file I/O functions because they are more portable and, in a few cases, slightly easier to use. However, for high-performance or multithread applications (depending on the actual implementation of your C compiler) you should rely on the OS/2 file services.

One final point: OS/2 allows you to bypass the logical structure of the disk and access the disk directly. Direct control of the disk hardware is beyond the scope of this book, however, and generally you will access the disk directly only when creating special disk utility programs, such as a file recovery program.

Table 6-1. The File I/O Subsystem Services

Service	Function
DosBufReset	Flushes the buffers associated with a file
DosChdir	Changes the current directory
DosChgFilePtr	Changes the location of the file pointer
DosClose	Closes a file
DosDelete	Deletes a file
DosDupHandle	Duplicates a file handle
DosFileLock	Locks a file
DosFindClose	Closes a directory search file handle
DosFindFirst	Finds the first file in the directory that matches the specified file name
DosFindNext	Finds the next file in the directory that matches the specified file name
DosMkdir	Makes a subdirectory
DosMove	Renames a file
DosNewSize	Resizes a file
DosOpen	Opens a file
DosPhysicalDisk	Returns information about the disk system
DosQCurDir	Returns information about the current directory
DosQCurDisk	Returns information about the current disk
DosQFHandState	Returns information about a file's handle
DosQFileInfo	Returns information about a file
DosQFileMode	Returns information about a file's mode
DosQFSInfo	Returns information about the file system
DosQHandType	Returns a handle's type
DosQVerify	Returns the state of the verify flag
DosRead	Reads data from a file
DosReadAsync	Reads data from a file but returns immediately
DosRmdir	Removes a subdirectory
DosScanEnv	Looks for a specified environmental variable
DosSearchPath	Searches for a file name given a path
DosSelectDisk	Changes the default drive
DosSetFHandState	Sets a file handle's state
DosSetFileInfo	Changes information associated with a file
DosSetFileMode	Changes a file's mode
DosSetFSInfo	Changes the file system information
DosSetMaxFH	Sets the maximum number of file handles
DosSetVerify	Changes the state of the verify flag
DosWrite	Writes data to a file
DosWriteAsync	Writes data to a file but returns immediately

FILE HANDLES

The OS/2 file subsystem operates on files through a file's handle, which is obtained when the file is first opened. Like all other OS/2 handles, a file handle is a 16-bit unsigned integer. You must obtain a valid file handle before attempting to use any of the file I/O services. You obtain the handle either by making a successful call to **DosOpen** or by using one of the built-in handles discussed later.

FILE POINTERS

All open disk files have associated with them a *file pointer*, which is used to keep track of the location in the file that is currently being accessed. OS/2 automatically maintains the file pointer during read or write operations. The file pointer is an unsigned long integer. For example, if a file is 100 bytes long and your program has just read the first 50 bytes, the value of the file pointer is 50. You can also set the value of the file pointer to reach a specific point in the file.

DosOpen AND DosClose

Before you can access a file you must obtain a handle to it. To do this you use the **DosOpen** service, which has the prototype

```
unsigned DosOpen(char far *filename,
                 unsigned short far *fhandle,
                 unsigned far *action, unsigned long size,
                 unsigned attr, unsigned openflags,
                 unsigned mode, unsigned long reserved);
```

where *filename* must be a null-terminated string that contains a valid path and file name for the file to be opened. The *fhandle* parameter points to the integer that contains the file's handle on return from a successful call.

The *action* parameter points to a set of flags encoded into an integer, which holds the action taken by a successful **DosOpen**. If the call fails, the value pointed to by *action* has no meaning. The *action* value will be one of the following.

Value	Meaning
1	File existed
2	File was created
3	File existed and its length was truncated

The *size* parameter specifies an initial length in bytes for a new or truncated file. The value can be 0. This parameter has no effect on a file that is opened for read operations.

The value of the *attr* parameter determines a file's attributes. It applies only to newly created files. The value of *attr* can be any valid (*i.e.*, not mutually exclusive) combination of the following:

Value	Type of File
0	Normal
1	Read-only
2	Hidden
4	System
20H	Archive

The value of the *openflags* parameter determines what action **DosOpen** takes depending on whether the specified file exists. Its value can be a combination of these values:

Value	Action
0	Returns error if the specified file already exists; otherwise, opens the file and returns success
1	Opens the file if it exists and returns success; otherwise, returns error
2	Opens an existing file, but truncates it; otherwise, returns error
10H	If specified file does not exist, creates it and returns success; if the file does exist, returns error

For example, if you wish to open a file that exists or create it if it does not, you would use a value of 11H (1 plus 10H).

The value of the *mode* parameter must specify both the access and the share mode of a file that is being created. All files can be accessed in one of three different ways: read-only, write-only, or read-write. For a single-tasking operating system, these access codes fully describe how access to the file is allowed. In OS/2, a multitasking system, the access

mode of a file is not sufficient to describe the file fully because it does not take into account the possibility of two or more processes attempting to access the file at the same time. To handle this situation all OS/2 files also have associated with them a *share attribute*, which is one of the following:

Share Attribute	Meaning
Deny write sharing	Only the process that opened the file can write to it, but other processes can read from it.
Deny read sharing	Only the process that opened the file can read from it, but other processes can write to it.
Deny read-write sharing	Only the process that opened the file can read or write to the file; all other processes are barred access.
Deny none	Any process can access the file at any time, in any way.

In addition to the access and file-sharing specifics, OS/2 lets you control a few other aspects of the file system. You can control the setting of the inheritance flag, which determines whether a child process inherits a file handle from the parent. You can tell the file system to return all I/O errors to the calling routine rather than the system-critical error routine. You can tell OS/2 that you do not want write operations to return until the information being written is actually put on the physical device (not simply written to a buffer). Finally you can specify that the drive is being accessed directly on a sector-by-sector basis, bypassing the disk's logical structure.

The values for the access, file-sharing, and miscellaneous flag settings are shown in Table 6-2. You combine the attributes you want to create the value desired for the *mode* parameter. (To combine the values, you simply add them together.) For example, to open a file for read-write operations with no sharing, use 12H.

The *reserved* parameter must be 0.

When the file is first opened, the file pointer is set to the beginning of the file and has the value 0.

Unless the write-to-device flag has been set, the OS/2 file system writes output to a buffer, not the actual physical file, until the buffer is full. Virtually all operating systems buffer disk input and output by

Table 6-2. File Mode Values

Type	Value	Meaning When Specified
Access	0	Read-only file.
	1	Write-only file.
	2	Read-write file.
Share	10H	Deny read-write sharing.
	20H	Deny write sharing.
	30H	Deny read sharing.
	40H	No access denied.
Inheritance	80H	File handles not passed on to child process.
Error	2000H	Immediate return to caller on error.
Write-to-device	4000H	Services that perform write operations do not return until information is written to the specified physical device.
Direct-access	8000H	Signals the system that direct device access will take place.

even multiples of a sector. When your program requests information, for example, the file system automatically reads a full sector even if only a partial one is needed. Subsequent sequential read requests can then obtain information from the buffer without waiting for a slow disk access. Output data is also buffered until a full sector can be written to disk, thus bypassing a number of time-consuming disk operations, each writing just a few bytes. Using the buffered approach to improve performance is not unique to OS/2. However, you must ensure that the contents of the buffer have been written to the file before your program terminates or before the handle associated with that file is destroyed. Because of the finite number of file handles available in the file system (20 by default), you also need some way to release a file handle for reuse when you are done with a file. To accomplish these goals OS/2 provides the **DosClose** service, whose prototype is

```
unsigned DosClose(unsigned short fhandle);
```

where *fhandle* must be a previously acquired file handle.

Before you can develop any meaningful examples using **DosOpen** and **DosClose**, you need to learn about **DosWrite**, the subject of the next section.

DosWrite

To write information to a file use the **DosWrite** service, which has the prototype

```
unsigned DosWrite(unsigned short fhandle, void far *buf,
                  unsigned count,
                  unsigned far *num_bytes_written);
```

The *fhandle* parameter must be a valid, previously obtained, file handle. The region pointed to by *buf* holds the information to be written to the file. The *count* parameter specifies the length of the buffer, or more properly the number of bytes in the buffer that should be written to the file. Finally, the *num_bytes_written* parameter points to an integer that contains the number of bytes actually written on return from the call. If an error occurs and it is not possible to write all the bytes requested, the value returned in the integer pointed to by *num_bytes_written* is different from the number requested.

OS/2 file operations are binary in nature and no character translations take place. (What you write is what you get!) The file system performs no formatting and is byte oriented by nature. That is, if you wish to write data other than characters, you must treat the data as a group of bytes. There is no OS/2 service that writes floating point numbers directly, for example. (You will see how to write other types of data later in this chapter.)

Each time you write to a file, its pointer is automatically advanced by the number of bytes written.

A SIMPLE FIRST EXAMPLE

To see how **DosOpen**, **DosWrite**, and **DosClose** work together, examine the following program that creates a new disk file called TEST.TST and writes the line "Hello OS/2 World!" to it. (The file must not exist. If it does, the call to **DosOpen** will fail.)

```
/* This programs writes output to a disk file. */

#define INCL_DOS

#include <os2.h>

main()
{
  unsigned short fh;
  unsigned action;
  unsigned count;
  char buf[80];

  strcpy(buf, "Hello OS/2 World!");

  /* create the file, no file sharing */
  if(DosOpen((char far *) "test.tst", /* filename */
          (unsigned short far *) &fh, /* pointer to handle */
          (unsigned far *) &action,   /* pointer to result */
          0L,     /* 0 length */
          0,      /* normal file */
          0x10,   /* create */
          0x11,   /* write-only, no-share */
          0L))    /* reserved */
  {
    printf("error in opening file");
    exit(1);
  }

  /* write a short message to it */
  if(DosWrite(fh, (void far *) buf, (unsigned) strlen(buf),
           (unsigned far *) &count))
    printf("error in write operation");

  /* close the file */
  if(DosClose(fh)) printf("error closing file");
}
```

The first time you run this program, it creates the file called TEST.TST and writes output to it. If you try to run the program a second time, however, OS/2 displays an "error in opening file" message, because the value of the *openflags* parameter specifies that the file will be created only if it does not exist.

Notice that this program checks for error returns from **DosOpen**, **DosWrite**, and **DosClose**. Errors are very common when you are dealing with files. One frequent error is failure to put a diskette into the drive. Another is running out of space on a disk. You must check for errors whenever you open a file or write to it. (Remember that closing a file may involve a write operation if a buffer must be written to disk. Hence **DosClose** must also be checked.) Unlike the screen or keyboard services, in which most of the functions are more-or-less guaranteed successful—and error checking can generally be ignored—many of the file system services have a significant likelihood of failure due to

uncontrollable circumstances. You simply must check for errors and take appropriate action if one occurs.

A Variation

As mentioned, the program just shown works only if the file does not already exist. You can change the value of the *openflags* parameter so that the file will be opened if it already exists or created if it doesn't. This can be accomplished by using the value 11H. This version of the program is shown here:

```
/* This program opens or creates a file and then
   writes output to it.
*/
#define INCL_DOS

#include <os2.h>
main()
{
  unsigned short fh;
  unsigned action;
  unsigned count;
  char buf[80];

  strcpy(buf, "Hello OS/2 World, again!");

  /* open or create the file, no file sharing */
  if(DosOpen((char far *) "test.tst", /* filename */
          (unsigned short far *) &fh, /* pointer to handle */
          (unsigned far *) &action,   /* pointer to result */
          0L,     /* 0 length */
          0,      /* normal file */
          0x11,   /* open or create */
          0x11,   /* write-only, no-share */
          0L))    /* reserved */
  {
    printf("error in opening file");
    exit(1);
  }

  /* write a short message to it */
  if(DosWrite(fh, (void far *) buf, (unsigned) strlen(buf),
           (unsigned far *) &count))
    printf("error in write operation");

  /* close the file */
  if(DosClose(fh)) printf("error closing file");
}
```

When you run this program, it opens an existing TEST.TST file and writes the new message to it, overwriting any existing contents. (Later you will see how to append information to a file.) If TEST.TST does not exist, it is created.

Buffer Lengths

As stated earlier, OS/2 buffers file information. At the time of this writing, its internal buffers are in even multiples of 512. For **DosWrite** to be as efficient as possible, it is best to call it with your own data buffers also in even multiples of 512. Of course if your application does not make this feasible, you can call **DosWrite** with data buffers of any value between 1 and 65,536.

DosRead

To read information from a file you use the **DosRead** service, which has the prototype

```
unsigned DosRead(unsigned short fhandle, void far *buf,
                 unsigned count, unsigned far *num_read);
```

The *fhandle* parameter is a valid, previously obtained file handle associated with the file from which you wish to read. The region pointed to by *buf* receives the information read. The value of *count* determines how many bytes are read from the file. The buffer receiving them must be at least *count* bytes long. The value pointed to by *num_read* will contain the number of bytes actually read after the call returns. The number of bytes requested and the number of bytes actually read may differ either because the end of the file has been reached or because an error has occurred.

OS/2 automatically updates the file pointer after each read operation.

The following program reads and displays the contents of a text file. You must specify the name of the file on the command line.

```
/* This program displays an entire file. */

#define INCL_DOS

#include <os2.h>

main(int argc, char *argv[])
{
  unsigned short fh;
  unsigned action;
```

```
    unsigned num_bytes;
    char buf[513];

    if(argc!=2) {
      printf("Usage: read <filename>\n");
      exit(1);
    }

    /* open the file, no file sharing */
    if(DosOpen((char far *) argv[1], /* filename */
            (unsigned short far *) &fh, /* pointer to handle */
            (unsigned far *) &action,   /* pointer to result */
            0L,     /* 0 length */
            0,      /* normal file */
            0x1,  /* open */
            0x10,  /* read-only, no-share */
            0L))   /* reserved */
    {
      printf("error in opening file");
      exit(1);
    }

    do {
      if(DosRead(fh, (char far *) buf, 512,
                 (unsigned far *) &num_bytes)) {
        printf("error reading file");
        exit(1);
      }
      buf[num_bytes] = '\0'; /* null terminate the buffer */
      printf(buf);
    } while(num_bytes);

    if(DosClose(fh)) printf("error closing file");
}
```

As this program illustrates, the easiest way to know when you have reached the end of the file is when the value of the *num_bytes* parameter is zero. The **DosRead** function does not return an EOF character.

One thing to notice about this program is that the buffer used to hold the data is one byte longer than the number of bytes requested to be read. In this situation the buffer must be transformed into a null-terminated string so that it can be used as a parameter to **printf()**. Not every application requires this step, of course.

One final point: As was the case with **DosWrite**, the **DosRead** service is most efficient when used with buffer lengths of even multiples of 512, although other values are perfectly valid.

RANDOM ACCESS

The OS/2 file system supports byte-addressable random access through the **DosChgFilePtr** service, which has the prototype

```
unsigned DosChgFilePtr(unsigned short fhandle,
                       long distance, unsigned origin,
                       unsigned long far *loc);
```

The *fhandle* parameter must contain a valid, previously obtained file handle. The **DosChgFilePtr** service works only on actual disk files and cannot be used with other devices. The value of *distance* determines how far, in bytes, the file pointer is to be moved relative to the origin. This is a signed value and may be either positive or negative. The value of origin determines how the value of *distance* is interpreted, as shown here:

Origin	Effect
0	Moves specified number of bytes from the start of the file
1	Moves specified number of bytes from the current location
2	Moves specified number of bytes from the end of the file

The value pointed to by *loc* holds the current value of the file pointer on return.

The following program uses the **DosChgFilePtr** service to let you scan a text file both forward and backward. You must specify the name of the file on the command line. The program supports these commands:

Command	Meaning
S	Go to beginning of the file.
E	Go to the end of the file.
B	Go back 512 bytes.
F	Go forward 512 bytes.
Q	Quit.

When the program begins execution, the first 512 bytes of the file are shown.

```
/* A file browse program. */

#define INCL_DOS

#include <os2.h>
```

```
main(int argc, char *argv[])
{
  unsigned short fh;
  unsigned action;
  unsigned num_bytes;
  long pos, p;
  char buf[513], ch;

  if(argc!=2) {
    printf("Usage: read <filename>\n");
    exit(1);
  }
  /* open the file, no file sharing */
  if(DosOpen((char far *) argv[1], /* filename */
          (unsigned short far *) &fh, /* pointer to handle */
          (unsigned far *) &action,   /* pointer to result */
          0L,     /* 0 length */
          0,      /* normal file */
          0x1,  /* open */
          0x10,  /* read-only, no-share */
          0L))   /* reserved */
  {
    printf("error in opening file");
    exit(1);
  }

  /* main loop */
  pos = 0L;
  do {
    if(DosRead(fh, (char far *) buf, 512,
               (unsigned far *) &num_bytes)) {
      printf("error reading file");
      exit(1);
    }
    buf[num_bytes] = '\0'; /* null terminate the buffer */
    printf(buf); /* display the buffer */

    /* see what to do next */
    ch = tolower(getch());
    switch(ch) {
      case 'e': /* move to end */
        DosChgFilePtr(fh, -512L, 2, (unsigned long far *) &pos);
        break;
      case 's': /* move to start */
        DosChgFilePtr(fh, 0L, 0, (unsigned long far *) &pos);
        break;
      case 'f': /* move forward */
        /* forward is automatic, so no change is required */
        pos = pos + num_bytes;
        break;
      case 'b': /* move backward */
        pos = pos - 512;
        if(pos<0L) pos = 0L;
        DosChgFilePtr(fh, pos, 0, (unsigned long far *) &p);
    }

  } while(ch != 'q');

  if(DosClose(fh)) printf("error closing file");
}
```

APPENDING TO A FILE

The way to add information to the end of a file is to advance the file pointer to the end of the file and then begin writing the new data. To accomplish this you could open the file for read-write operations and read to the end of the file. But this method is very inefficient. The best way to get to the end of the file is to use **DosChgFilePtr** in a statement like this:

```
DosChgFilePtr(fh, 0L, 2, (unsigned long far *) &pos);
```

This tells OS/2 to move the file pointer to the end of the file. The *2* in the *origin* parameter and the *0L* in the *distance* parameter ensure that the file pointer will be at the physical end of the file.

The following program uses this method to add lines of text entered at the keyboard to the file TEST.TST. To stop inputting lines, enter the word *quit* when prompted for the next line.

```
/* This program opens a file, reads lines from the keyboard
   and appends each line to the end of the file.
*/
#define INCL_DOS

#include <os2.h>

main()
{
  unsigned short fh;
  unsigned action;
  long pos;
  unsigned count;
  char buf[80];

  /* open or create the file, no file sharing */
  if(DosOpen((char far *) "test.tst", /* filename */
          (unsigned short far *) &fh, /* pointer to handle */
          (unsigned far *) &action,   /* pointer to result */
          0L,     /* 0 length */
          0,      /* normal file */
          0x11,   /* open or create */
          0x11,   /* write-only, no-share */
          0L))    /* reserved */
  {
    printf("error in opening file");
    exit(1);
  }

  /* go to the end of the file */
  DosChgFilePtr(fh, 0L, 2, (unsigned long far *) &pos);

  /* continue adding to the file until the word "quit" is
     entered
  */
```

```
  do {
    printf("enter message (<80 chars) message: ");
    gets(buf);
    if(DosWrite(fh, (void far *) buf, (unsigned) strlen(buf),
                (unsigned far *) &count))
      printf("error in write operation");
  } while(strcmp("quit", buf));
  /* close the file */
  if(DosClose(fh))  printf("error closing file");
}
```

READING AND WRITING OTHER DATA TYPES

You can use the OS/2 file system services to read and write data types other than characters (bytes) by treating a variable of a different type as a buffer and using its address and length in the calls to **DosRead** and **DosWrite**. (Remember that you can obtain the size of any data type by using the **sizeof** compile time operator.) For example, the following program first writes a double value to the file TEST.TST and then reads it back, displaying the value to the screen for verification:

```
/* This program illustrates how to write a double value to
   a file and read it back.
*/
#define INCL_DOS

#include <os2.h>

main()
{
  unsigned short fh;
  unsigned action;
  unsigned count;
  float dbl;
  unsigned long pos;

  dbl = 101.125;

  /* open or create the file, no file sharing */
  if(DosOpen((char far *) "test.tst", /* filename */
          (unsigned short far *) &fh, /* pointer to handle */
          (unsigned far *) &action,   /* pointer to result */
          0L,     /* 0 length */
          0,      /* normal file */
          0x11,   /* open or create */
          0x12,   /* read-write, no-share */
          0L))    /* reserved */
    {
    printf("error in opening file");
    exit(1);
  }

  /* write a double value to it */
```

```
  if(DosWrite(fh, (double far *) &dbl, sizeof(dbl),
           (unsigned far *) &count))
    printf("error in write operation");

  /* clear the dbl variable */
  dbl = 0.0;

  /* reset the file pointer to start of file */
  DosChgFilePtr(fh, 0L, 0, (unsigned long far *) &pos);

  if(DosRead(fh, (double far *) &dbl, sizeof(dbl),
            (unsigned far *) &count))
  {
    printf("error reading file");
    exit(1);
  }

  printf("%f", dbl); /* verify that info read correctly */

  /* close the file */
  if(DosClose(fh)) printf("error closing file");
}
```

You can use this same basic approach on more complex data types such as arrays, unions, and structures. Just be sure that you are passing the variable's address, not its value, to **DosRead** or **DosWrite**.

READING AND WRITING TO A DEVICE

The OS/2 file system lets you access certain devices as if they were files. For example, you can open the console (screen and keyboard) and then read and write to it. To open a device, use the device's name in place of a file name in the **DosOpen** call. The devices supported by OS/2 are

clock$	con	mouse$
com1	kbd$	nul
com2	lpt1	pointer$
com3	lpt2	prn
com4	lpt3	screen$

The most interesting of these are **com1** through **com4** (the serial communication ports) and **lpt1** through **lpt3** and **prn** (the printer ports).

One thing to keep in mind is that not all devices support all modes of operation. For example, if you open **screen$**, you may write to the screen but not read from it. As you will see, disk files also support random access operations, but devices do not.

This program opens the keyboard, reads a line of text, and displays the contents of the buffer:

```
/* This program reads input from the keyboard. */

#define INCL_DOS

#include <os2.h>

main()
{
  unsigned short fh;
  unsigned action;
  unsigned count;
  char buf[80];

  /* open the keyboard */
  if(DosOpen((char far *) "kbd$", /* keyboard */
          (unsigned short far *) &fh, /* pointer to handle */
          (unsigned far *) &action,   /* pointer to result */
          0L,     /* 0 length */
          0,      /* normal "file" */
          0x11,   /* open or create */
          0x10,   /* write-only, no-share */
          0L))    /* reserved */
    {
    printf("error accessing the keyboard");
    exit(1);
  }

  if(DosRead(fh, (void far *) buf, 80,
           (unsigned far *) &count))
     printf("error in read operation");

  printf(buf);

  if(DosClose(fh)) printf("error closing the keyboard");

}
```

The following program opens **lpt1** and writes a message to it:

```
/* This program writes output to the printer. */

#define INCL_DOS

#include <os2.h>
main()
{
  unsigned short fh;
  unsigned action;
  unsigned count;
  char buf[80];

  strcpy(buf, "Hello OS/2 World!");

  /* open or create the file, no file sharing */
  if(DosOpen((char far *) "lpt1", /* printer */
```

```
            (unsigned short far *) &fh, /* pointer to handle */
            (unsigned far *) &action,   /* pointer to result */
            0L,     /* 0 length */
            0,      /* normal "file" */
            0x11,   /* open or create */
            0x11,   /* write-only, no-share */
            0L))    /* reserved */
   {
     printf("error accessing the printer");
     exit(1);
   }

   if(DosWrite(fh, (void far *) buf, (unsigned) strlen(buf),
            (unsigned far *) &count))
      printf("error in write operation");

   if(DosClose(fh)) printf("error closing the printer");
}
```

Generally speaking, you will not use the OS/2 file system to write to the screen, read the keyboard or the mouse, or access the system clock. The OS/2 dedicated services that perform these functions will generally be faster than going through the file system. However, you should use the file system to access the printer ports because OS/2 can automatically route output to the proper port without your program needing intimate knowledge of the system's configuration.

THE OS/2 STANDARD DEVICES

OS/2 has three built-in file handles, which are associated with three standard devices. These handles are created when your program begins executing. The handles and their meaning are

Handle	Meaning
0	Standard input
1	Standard output
2	Standard error (output)

By default standard input is associated with the keyboard, and standard output and standard error are associated with the screen. However, because OS/2 supports I/O redirection of its standard devices, input and output can be routed to disk files or other devices.

The following program writes a message to standard output:

```
/* This program writes output to Standard Output. */
#define INCL_DOS

#include <os2.h>
```

```
main()
{
  unsigned count;
  char buf[80];

  strcpy(buf, "Hello OS/2 World!");

  /* write a short message to it */
  if(DosWrite(1, (void far *) buf, (unsigned) strlen(buf),
          (unsigned far *) &count))
    printf("error in write operation");
}
```

Notice that the program does not have to open standard output because OS/2 does so automatically when the program begins. The program does not close standard output because this, too, is performed automatically. If this program were called STDOUT, executing it using this command line causes the message to be written to the screen:

```
STDOUT
```

However, using the following command line causes the message output by the program to be written to a file called MESS.

```
STDOUT >MESS
```

DISPLAYING THE DIRECTORY

Application programs commonly need to display the contents of a directory so that the user can make a file selection. OS/2 makes this very easy to do through its **DosFindFirst** and **DosFindNext** services. Their prototypes are

```
unsigned DosFindFirst(char far *mask, unsigned short *handle,
                      unsigned attr,
                      FILEFINDBUF far *info,
                      unsigned buflength,
                      unsigned far *count,
                      unsigned long reserved);
unsigned DosFindNext(unsigned short handle,
                      FILEFINDBUF far* info,
                      unsigned buflength,
                      unsigned far *count);
```

For **DosFindFirst**, the *mask* parameter is a null-terminated string that holds the file name you are looking for. This string can include the * and ? wild card characters. A directory handle is returned in the variable pointed to by *handle*. This handle is used in subsequent calls to **DosFileNext**. Prior to the call to **DosFindFirst** *handle* must contain the value 1 or FFFFH. If its value is 1, OS/2 supplies a default handle. However, if you will be searching for more than one specific file, use FFFFH, which causes OS/2 to return a handle that can be used in subsequent calls to **DosFindNext**. The *attr* parameter specifies the type of file you are looking for. It can be any valid (not mutually exclusive) combination of the following values:

Value	File Type
0	Normal
1	Read-only
2	Hidden
4	System
10H	Subdirectories
20H	Archive

The structure of type **FILEFINDBUF** pointed to by *info* receives information about the file if a match is found. This structure type is defined as

```
typedef struct _ FILEFINDBUF {
        FDATE   fdateCreation; /* creation date */
        FTIME   ftimeCreation; /* creation time */
        FDATE   fdateLastAccess; /* last access date */
        FTIME   ftimeLastAccess; /* last access time */
        FDATE   fdateLastWrite; /* last write date */
        FTIME   ftimeLastWrite; /* last write time */
        unsigned long cbFile; /* file length */
        unsigned long cbFileAlloc; /* total space allocated */
        unsigned attrFile; /* file attribute */
        unsigned char cchName; /* filename length */
        char achName[13]; /* filename */
} FILEFINDBUF;
```

The types **FDATE** and **FTIME** are defined by Microsoft as

```
typedef struct __FTIME {
    unsigned twosecs : 5;
    unsigned minutes: 6;
    unsigned hours   : 5;
} FTIME;

typedef struct __FDATE {
    unsigned day     : 5;
    unsigned month   : 4;
    unsigned year    : 7;
} FDATE;
```

The *buflength* parameter specifies the length of the **FILEFINDBUF** structure. The integer pointed to by *count* specifies the number of matches to find and holds the number of matches found on return. It is generally best to give *count* a value of 1. If no match is found, 0 is returned. The *reserved* parameter must be 0.

The parameters for **DosFindNext** have the same meaning as those for **DosFindFirst**.

If you are looking for only one specific file and fully specify that file's name (no wild cards) in the call to **DosFindFirst**, you will not need to use **DosFindNext**. If you are searching for (potentially) several matches, however, the basic method is to call **DosFindFirst** to obtain the first match (if any) and a directory handle to use in subsequent calls to **DosFindNext**.

There are two ways to determine when the last match has been found:

1. Both **DosFindFirst** and **DosFindNext** fail and return an error code if no match is found.
2. The *count* parameter is zero when no (more) matches are found.

The following program lists the current working directory. It displays the file's name and length.

```
/* This program lists the directory. */

#define INCL_DOS

#include <os2.h>

void show_dir(void);

main()
{
  show_dir();
}

/* Display the directory. */
void show_dir()
{
  FILEFINDBUF f;
  unsigned short hdir;
  unsigned count;

  hdir = 0xffff; /* cause a new handle to be returned */
  count = 1;  /* find the first match */
  DosFindFirst((char far *) "*.*", (unsigned short far *) &hdir,
               0x0, (FILEFINDBUF far *) &f, sizeof(f),
               (unsigned far *) &count, 0L);
  do {
    printf("%-13s %d\n", f.achName, f.cbFile);
    DosFindNext(hdir, (FILEFINDBUF far *) &f, sizeof(f),
                (unsigned far *) &count);
  }while(count);
  DosFindClose(hdir);
}
```

ACCESSING INFORMATION ABOUT THE DISK SYSTEM

It is not uncommon for an application to need to have knowledge of various pieces of information about the disk system, including such things as the total free storage, the number of bytes per sector, or the number of sectors per cluster. To obtain this information, OS/2 supplies the **DosQFSInfo** service, which has the prototype

```
unsigned DosQFSInfo(unsigned drive, unsigned info-type,
                    char far *info, unsigned buflength);
```

where *drive* specifies the number of the drive you want to receive information about. If it is 0, the default drive is used. Otherwise, use 1 for drive A, 2 for drive B, and so on. The *info-type* parameter specifies what type of information will be returned. If it is 1, on return *info*

points to a structure of type **FSALLOCATE**, which is defined as

```
typedef struct _FSALLOCATE {
        unsigned long idFileSystem; /* system identifier */
        unsigned long cSectorUnit; /* sectors per cluster */
        unsigned long cUnit; /* total number of sectors */
        unsigned long cUnitAvail; /* available sectors */
        unsigned cbSector; /* bytes per sector */
} FSALLOCATE;
```

In some OS/2 literature, a cluster is called a *unit*, but this book will continue to use the term *cluster* because it is more common.

If *info-type* is 2, *info* points to a structure of this type:

```
type struct _FSALLOCATE2 {
        FDATE fdateCreation; /* creation date of volume label */
        FTIME ftimeCreation; /* creation time of volume label */
        char cchName;        /* length of volume name */
        char achName[14];    /* volume name */
} FSALLOCATE2;
```

Note: FSALLOCATE2 is not defined in any header files provided by Microsoft and must be defined explicitly by your program. (This situation could change. If you are using a different compiler, this structure could also be defined in a header file provided with your compiler.)

This program displays the number of bytes per sector, the number of sectors per cluster, the total disk space, and the total free disk space for the default drive. The total disk space is computed by multiplying the number of bytes per sector by the number of sectors per cluster by the number of clusters on the disk. The free space is computed by multiplying the number of bytes per sector by the number of sectors per cluster by the number of clusters available.

```
/* Demonstrate the DosQFSInfo service and display the number
   of bytes per sector, sectors per cluster, total disk space,
   and available disk space.
*/

#define INCL_DOS
```

```
#include <os2.h>

main()
{
  FSALLOCATE f;

  DosQFSInfo(0, 1, (char far *) &f,
             sizeof f);

  printf("Bytes per sector: %ld\n", f.cbSector);
  printf("Sectors per cluster: %ld\n", f.cSectorUnit);
  printf("Total disk space: %ld\n",
         f.cbSector * f.cSectorUnit * f.cUnit);
  printf("Total available disk space: %ld\n",
         f.cbSector * f.cSectorUnit * f.cUnitAvail);
}
```

EXAMINING AND CHANGING THE DIRECTORY

OS/2 provides two important directory services called **DosQCurDir** and **DosChgDir**, which are used to return the path name of the current directory and to change the current directory. Their prototypes are shown here:

```
unsigned DosQCurDir(unsigned drive, char far *path,
                    unsigned far *size);
unsigned DosChDir(char far *path, unsigned long reserved);
```

For **DosQCurDir** the *drive* parameter specifies the drive to be operated on. To use the default drive, use 0 for *drive*. For drive A, use 1, for drive B use 2, and so on. Upon return the character array pointed to by *path* holds the path name of the directory. The integer pointed to by *size* must hold the length of the array pointed to by *path* prior to the call, and it returns the length of the path name.

For **DosChDir**, *path* points to the character that holds the new directory path name. The *reserved* parameter must be 0L.

This program displays the current directory name, switches to the root directory, and then switches back to the original directory:

```
/* Displaying and changing the directory. */

#define INCL_DOS

#include <os2.h>

main()
```

```
{
  char olddirname[64], newdirname[64];
  unsigned size;

  size = 63;

  DosQCurDir(0, (char far *) olddirname, (unsigned far *) &size);
  printf("current directory: %s\n", olddirname);
  DosChdir("\\", 0L);
  DosQCurDir(0, (char far *) newdirname, (unsigned far *) &size);
  printf("current directory: %s\n", newdirname);
  DosChdir(olddirname, 0L);
  DosQCurDir(0, (char far *) newdirname, (unsigned far *) &size);
  printf("current directory: %s\n", newdirname);

}
```

7

AN INTRODUCTION TO MULTITASKING

The preceding chapters in this book have covered some very important OS/2 API services. If you are moving from DOS or another single-tasking system to OS/2, the material in the previous chapters, although necessary, is nothing especially new or exciting. However, this chapter introduces you to OS/2's multitasking capabilities, in which much of its power lies. If you are new to a multitasking environment, this is where the real fun begins!

The use of multitasking can dramatically increase the efficiency of most applications. For example, in a software development situation multitasking allows you to edit, compile, and test simultaneously. Part of a word processor program can be inputting text, while another part is formatting it for printing, and yet another part is actually printing the document. The entire point of a multitasking, single-user system like OS/2 is to help the user achieve greater throughput by minimizing needless idle time.

This chapter covers some of the basic OS/2 multitasking services. The next chapter builds on the material presented here and discusses interprocess and interthread communication and synchronization issues. The time you invest in understanding the concepts presented here will really pay off later.

As you will recall, OS/2 implements multitasking on both a process and a thread level. Hence, OS/2 provides two sets of multitasking services: one to create and support processes and one to create and support threads. This chapter looks at both, beginning with processes.

A WORD OF WARNING

Before we begin it is important to emphasize one important point: You must never make any assumptions about the way multitasking routines will be executed by OS/2. You must never assume that one routine will execute before another or that it will execute for a given number of milliseconds. For example, if you need one multitasked routine to execute before another, perhaps to initialize something, you must explicitly design this into your program. If you find, through experimentation, that one multitasked routine always executes before another, it is not acceptable to use this "fact" in your program, for three important reasons:

1. Future versions of OS/2 may schedule tasks differently. (Actually, nothing in the OS/2 documentation says that you can assume anything about the way OS/2 schedules tasks even within the same version.)
2. In the future, OS/2 may be designed to run on a multiple-CPU computer, thus allowing true concurrent execution of multiple tasks. In that case two tasks that might have been sequenced in a single CPU system will be run simultaneously.
3. Future versions of OS/2 may change the way time slices are allocated, causing the "first" routine to begin execution but not finish before the "second" begins.

Remember that when you are dealing with multitasked routines there is no valid concept of which routine is executed "first," unless you have explicitly provided for this in your program.

To write solid multitasked code you must assume that all multitasked routines are actually executed simultaneously, whether they are in your current environment or not. Most of the troubles you will experience when you use multitasking inside your programs will be caused by forgetting this important point.

PROCESSES VERSUS THREADS

The distinction between a process and a thread, covered earlier in this book, is summarized here. A thread is a dispatchable piece of code; that is, the OS/2 scheduler executes threads. A thread does not own resources. A process consists of at least one thread and may have several. A process owns resources. Very loosely, a process is a program and a thread is like a subroutine in that program.

MULTIPLE PROCESSES

OS/2 has nine services (shown in Table 7-1) that are used to oversee the creation and operation of multiple processes. As you can probably tell by looking at Table 7-1, OS/2 lets your program begin the concurrent execution of another program. The program that initiates the second program is called the *parent* and the program that it causes to be executed is called the *child*. A parent can create a child in two different ways:

1. It can simply cause the child to be run in the parent's session using **DosExecPgm**.
2. It can create another session and run the child under that session, in either an autonomous or a controlled mode, using **DosStartSession**.

Table 7-1. OS/2 Multiple Process Services

Service	Function
DosCWait	Waits for a child process to terminate
DosExecPgm	Loads and executes another process
DosExit	Terminates the current process
DosExitList	Registers functions to be called when the process terminates
DosGetPid	Returns a process's identification code
DosSelectSession	Makes specified session foreground
DosSetSession	Sets a session's status
DosStartSession	Starts a new session
DosStopSession	Stops a session

Most of the time when you want one program to cause the execution of another, related program, you use **DosExecPgm**. The main use for **DosStartSession** is at system initialization, when you might want to begin several sessions automatically.

We will begin with a look at **DosExecPgm** and its support functions.

DosExecPgm

To execute a second process from a currently executing program, use **DosExecPgm**, which has the prototype

```
unsigned DosExecPgm(char far *failbuf, unsigned failbuf_size,
                    unsigned exec_mode,
                    char far *args,
                    char far *env,
                    RESULTCODES far *result,
                    char far *filename);
```

The buffer pointed to by *failbuf* receives a message that helps explain the cause of any failure to execute the specified program successfully. The *failbuf_size* parameter specifies the size of the fail buffer. The *exec_mode* parameter specifies how the child program will be executed and must be one of these values:

Value	Meaning
0	Execute synchronously
1	Execute asynchronously and discard child's termination code
2	Execute asynchronously and save child's termination code
3	Execute in debug mode
4	Detach child

When the child program is executed synchronously, the parent suspends execution until the child has terminated, at which time the parent resumes. In a DOS environment, this is the only way that one program can run another. However, in OS/2's multitasking environment, synchronous execution is seldom used and is not of much interest. When the child is run asynchronously, the parent and the child execute concurrently. If the parent needs extensive information about how the child terminated, call **DosExecPgm** with the *exec_mode* set to 2;

if not, use the value 1. The debug mode is used for tracing. If you want to detach the child, use the value 4.

The parameters *args* and *env* point, respectively, to arrays that hold any command line arguments and environment variables required by the child process. Either or both may be null. The array pointed to by *args* begins with the null-terminated name of the program followed by a double-null-terminated list of the arguments. For example, if the child program's name is TEST and you want to pass it the argument "HELLO THERE", call **DosExecPgm** with *args* pointing to this string:

```
"TEST\0HELLO THERE\0\0"
```

The environment variables are passed to the child as null-terminated strings with the last being a double-null-terminated string.

The structure pointed to by *result* receives information about the termination of the child process. The structure is defined like this:

```
typedef struct _RESULTCODES {
        unsigned codeTerminate;
        unsigned codeResult;
} RESULTCODES;
```

If the child is executed asynchronously, **codeTerminate** holds the process identifier (PID) associated with the child process. For asynchronous execution, the **codeResult** field is not used. If the child is executed synchronously, **codeTerminate** will be 0 for normal termination, 1 for hardware error, 2 for system trap, and 3 if the process was killed. For synchronous execution, **codeResult** holds the child's exit code.

The array pointed to by *filename* contains the drive, path, and name of the program to be executed.

As with all the API services, **DosExecPgm** returns zero if successful and nonzero otherwise.

For example, this program first asynchronously executes a program called TEST.EXE and then begins printing 1s on the screen, sleeping a little each time through the loop, until you press a key.

```
/* This program asynchronously executes another. */
#define INCL_DOS

#include <os2.h>
```

```
main()
{
  char fail[128];
  RESULTCODES result;

  if(DosExecPgm((char far *) fail, 128,
            1, /* run asynchronous */
            (char far *) "", /* no command line args */
            (char far *) "", /* no environment args */
            (RESULTCODES far *) &result, /* result */
            "TEST.EXE"))  /* name of program */
     printf("exec error");

  do {
    printf("1 ");
    DosSleep(100L);
  } while(!kbhit());
}
```

Use this for the TEST.EXE program:

```
/* This is the TEST program used by several of the example
   multitasking programs in this chapter.
*/
#define INCL_DOS

#include <os2.h>

main()
{
  do {
    printf("2 ");
    DosSleep(11L);
  } while(!kbhit());
}
```

When both programs are executing you see a series of *1*s and *2*s displayed on the screen. Because of the difference in the **DosSleep** parameter, about four times as many *2*s are shown as *1*s. (You might want to try varying the sleep parameters to see the effect. This will give you insight into how the OS/2 scheduler works.) Notice that both programs check for a keystroke prior to termination. Since the key is not read by either program, the keybuffer is not cleared. Therefore, a single keypress terminates both programs. However, in real applications you need to make sure that input meant for one program is not accidentally routed to another program.

When the child begins executing, it inherits the parent's environment, including all open file handles (except those with the inheritance flag set to 0). The child can access these files without opening them. Of

course the parent's environment can be overridden or augmented by the contents of the environment array passed at the time of the **DosExecPgm** call.

With a slight modification to the **DosExecPgm** call in the parent program, the command line argument "HI" can be passed to the TEST.EXE program, as shown here:

```
if(DosExecPgm((char far *) fail, 128,
           1, /* run asynchronous */
           (char far *) "TEST\0HI\0\0", /* <== pass arg */
           (char far *) "", /* no environment args */
           (RESULTCODES far *) &result, /* result */
           "TEST1.EXE")) /* name of program */
   printf("exec failed");
```

This version of TEST.EXE prints the argument before proceeding:

```
#define INCL_DOS

#include <os2.h>

/* This time, show the command line argument */
main(int argc, char *argv[])
{
  printf("%s", argv[1]);
  do {
    printf("2 ");
    DosSleep(11L);
  } while(!kbhit());
}
```

Two important points to remember:

1. A parent can execute more than one child process.
2. A child process can execute its own child processes.

Waiting for a Child to Terminate When Using DosCWait

In multitasking environments it is not uncommon for the parent process at some point to wait until an asynchronous child process has finished. For example, a database program may initiate a sort process and then continue processing user input. However, the parent will have to wait until the sort is complete before processing a request to print the

database. In other words, it is very common for a parent and an asynchronously executing child process to execute concurrently until some special event causes the parent to wait for the child to finish. This differs from simple synchronous execution in which the parent and child never execute concurrently. To allow the parent to wait for a child, OS/2 includes the **DosCWait** service, whose prototype is

```
unsigned DosCWait(unsigned descendants,
                  unsigned wait,
                  RESULTCODES far *results,
                  unsigned far *Tpid,
                  unsigned pid);
```

The *descendants* parameter specifies whether **DosCWait** should wait for the termination of just the specified process or of the specified process and all (if any) of its child processes. If *descendants* is 0, the parent waits only for the specified process. If it is 1, the parent waits for the specified process and any of its children.

The *wait* parameter specifies whether **DosCWait** actually waits for the specified process to terminate or simply returns immediately. If its value is 0, the parent waits for the process to terminate. If it is 1, the parent returns immediately with the result codes of an already terminated process. (However, if the specified process is still executing when **DosCWait** is called with the no-wait option, it returns an error message.)

The structure pointed to by *result* is of type **RESULTCODES** and is the same as that described earlier in the discussion of **DosExecPgm**.

The variable pointed to by *Tpid* will hold the process identifier of the terminating process as set by **DosCWait**.

The *pid* parameter specifies the process identifier of the process to wait for. If it is null, the first child process to terminate causes a return and the process identifier of this child is loaded into the *Tpid* parameter. Otherwise, **DosCWait** waits only for the specified process. If the specified process does not exist, **DosCWait** returns an error message.

The following program executes the TEST.EXE program shown earlier and waits for it to end. (To end the TEST.EXE program, press any key.)

```
/* This program demonstrates the DosCWait service. */
#define INCL_DOS
```

```
#include <os2.h>

main()
{
  char fail[128];
  RESULTCODES result, waitresult;
  unsigned proc;

  if(DosExecPgm((char far *) fail, 128,
            1, /* run asynchronous */
            (char far *) "", /* no command line args */
            (char far *) "", /* no environment args */
            (RESULTCODES far *) &result, /* result */
            "TEST.EXE")) /* name of program */
    printf("exec error");

  DosCWait(0,  /* wait for specified process only */
           0,  /* wait for termination */
           (RESULTCODES far *) &waitresult, /* result */
           (unsigned far *) &proc, /* PID */
           result.codeTerminate); /* PID to wait on */

  printf("child process terminated\n");
}
```

Notice how the process identifier of TEST.EXE is first returned in the **result.codeTerminate** field by **DosExecPgm** and then used by **DosCWait** to specify the specific process to wait for.

It is important to understand that when **DosCWait** is called using its wait mode, the calling process is suspended, thus freeing the CPU.

Killing a Process

The parent can terminate a child process. To understand why this is necessary, imagine that you have created a large relational database system. The main (parent) process includes all the user input and query functionality. To achieve uninterrupted use, however, you allocate time-consuming tasks such as printing, sorting, mail merges, and backups to separate child processes that are executed only when needed. In such a system, it is very likely that from time to time you will need to terminate one or more child processes because they are no longer needed. To accomplish this task OS/2 provides **DosKillProcess**, which has the prototype

```
unsigned DosKillProcess(unsigned descendants,
                        unsigned pid);
```

If the *descendants* parameter is 0, the specified process and any descen-

dants are killed. If it is 1, only the specified process is terminated. The *pid* parameter is the process identifier for the process to be stopped.

DosKillProcess can fail and return nonzero only if the specified process does not exist.

To see **DosKillProcess** in action, try this program, which executes the TEST.EXE program, waits 5000 milliseconds, and then kills it. Try the program two ways:

1. Simply do nothing, letting it kill TEST.EXE. In this case, OS/2 prints the message "child process terminated."
2. After TEST.EXE begins execution but before it is killed by its parent, press any key. (Remember, TEST.EXE terminates if you press a key.) In this case, when the parent tries to kill it with the **DosKillProcess**, it fails and the message "child process already terminated" appears.

```
/* This program executes a second program, waits a while and
   then kills the second program.
*/

#define INCL_DOS

#include <os2.h>

main()
{
  char fail[128];
  RESULTCODES result, waitresult;
  unsigned proc;

  if(DosExecPgm((char far *) fail, 128,
             1, /* run asynchronous */
             (char far *) "", /* no command line args */
             (char far *) "", /* no environment args */
             (RESULTCODES far *) &result, /* result */
             "TEST.EXE")) /* name of program */
    printf("exec error");

  DosSleep(5000L);

  /* kill child */
  if(DosKillProcess(1, result.codeTerminate))
    printf("child process already terminated");
  else
    printf("child process terminated\n");
}
```

Creating an Exit Function List

Since a parent function can terminate a child process unexpectedly, it may be advisable to ensure that the child has some means of dying a clean death. For example, you will want the child program to flush any disk buffers and close all files. Special hardware devices may need to be reset, and it may even be appropriate to notify the user that the process is being killed. To enable the child to perform these tasks, OS/2 calls a special list of functions whenever a process (child or parent) terminates. The functions that comprise this list are called *exit functions*. Collectively they are called the *exit function list*. OS/2 provides the **DosExitList** service to support the exit function list. Its prototype is

```
unsigned DosExitList(unsigned operation,
                     void far *exfunc(unsigned term_code));
```

The value of *operation* determines what **DosExitList** does. The valid values are shown here*:

Value	Meaning
1	Add a function to the exit list
2	Remove a function from the exit list
3	Current exit function is done; move on to the next function in the exit list

To add or remove a function from the list, you must pass a pointer to the function in the *exfunc* parameter. The function must be declared as follows:

```
void far func(unsigned term_code);
```

The function will be passed a termination code in the *term_code* parameter, which will be one of these values*:

Value	Meaning
0	Normal termination
1	Unrecoverable error
2	System trap error
3	Process killed

*These table's were adapted from tables in *Operating System/2 Programmer's Reference Manual*, with permission of Microsoft Corporation.

Your exit function can take different actions based on the termination code if so desired.

The basic approach to establishing an exit function is first to call **DosExitList** to add the function to the list. At termination the last thing your function must do is call **DosExitList** with the *operation* parameter set to 3, to move to the next function in the list. If for some reason you want to remove a function that you previously added to the list, call **DosExitList** with *operation* set to 2.

There is one very important thing to remember about an exit function: It cannot be terminated by OS/2. This means that your exit functions should be very short and never, under any circumstances, delay the termination of the process more than a few milliseconds. Because the environment surrounding the exit functions is dying, it is imperative that your function does what it needs to do as quickly as possible. An incorrectly constructed exit function cannot crash OS/2, but it can make it impossible for OS/2 to complete its termination of the process and thereby degrade system performance.

Another important point: You cannot assume that two or more exit functions will always be called in the same order. OS/2 guarantees to call them, but not in any special sequence.

As a simple example, this program puts the function **exfunc()** into the exit list and then prints 1000 numbers. On termination, the **exfunc()** function displays whether the process terminated normally or was killed by your pressing CTRL-C.

```
/* This program creates an exit function, exfunc(), which
   is called when the program terminates.
*/

#define INCL_DOS

#include <os2.h>

void far exfunc(unsigned);

main()
{
  int i;

  DosExitList(1, exfunc);
  for(i=0; i<1000; i++) printf("%d ");
}

/* This function is automatically called at termination. */
```

```
void far exfunc(unsigned term_code)
{
  if(term_code==0)
    printf("program terminating normally");
  else
    printf("program terminating abnormally");

  /* done with this exit function, move on */
  DosExitList(3, (void far *) 0);
}
```

Error Checking

A wide variety of errors can occur when you create or manipulate processes. For example, in a given situation OS/2 may not be able to create a new process because all process identifiers are already allocated. It is important to watch for errors in your applications and take appropriate action if one occurs.

CREATING NEW SESSIONS

When you used **DosExecPgm** to start new processes, these new processes ran in the same session (sometimes called a *screen group*) as the parent. Although this is very useful for related processes that interact with each other to form a unit, it is not very desirable when the processes are not related. However, OS/2 allows you to start a process in its own session by using the **DosStartSession** service, whose prototype is

```
unsigned DosStartSession(STARTDATA far *sdata,
                         unsigned far *sid,
                         unsigned far *pid);
```

The structure pointed to by *sdata* is defined like this:

```
typedef struct _STARTDATA {
        unsigned cb;          /* size of struct */
        unsigned Related;     /* session related to parent */
        unsigned FgBg;        /* foreground or background */
        unsigned TraceOpt;    /* trace active? */
        char far *PgmTitle;   /* session title */
```

```
        char far *PgmName; /* name of program to execute */
        char far *PgmInputs; /* command line args */
        char far *TermQ;     /* termination queue or null */
} STARTDATA;
```

The **cb** field must hold the length of the **STARTDATA** structure. If **Related** is 0, the new session is completely independent of the parent. If it is 1, the new session is a child of the parent. If **FgBg** is 0, the new session becomes the foreground task; if it is 1, the new session becomes a background task. The new session can become a foreground task only if the parent is in foreground when it creates the session. If **TraceOpt** is 0, the new session is not set up for tracing; if it is 1, the new session can be traced. The string pointed to by **PgmTitle** is the name of the session and may be null. The string pointed to by **PgmName** is the name of the program that will begin running in the new session. The string pointed to by **PgmInputs** contains any command line arguments needed by the program and may be null. The string pointed to by **TermQ** is the name of the termination queue and may be null.

The *sid* parameter points to a variable that receives the session identifier when the call returns. The *pid* parameter points to a variable that receives the process identifier of the process run in the newly created session.

This program begins a new session called "my session" and starts running the TEST.EXE program. When you try this program, remember that you need to have TEST.EXE in the current working directory.

```
/* Start a new session and run the TEST.EXE program. */

#define INCL_DOS

#include <os2.h>

main()
{
  STARTDATA d;
  unsigned sid, pid;

  d.cb = sizeof(d); /* size of struct */
  d.Related = 0;    /* not related */
  d.FgBg = 0;       /* foreground */
  d.TraceOpt = 0;   /* no tracing */
  d.PgmTitle = (char far *) "my session"; /* session name */
```

```
    d.PgmName = (char far *)"c:\\pm\\test.exe"; /* name */
    d.PgmInputs = (char far *) "";  /* no command line args */
    d.TermQ = (void far *) 0; /* no queue */

    DosStartSession((STARTDATA far *) &d,  /* session data */
                    (unsigned far *) &sid, /* session id */
                    (unsigned far *) &pid);/* process id */
}
```

In this program the new session is not a child of the parent and becomes the foreground task.

Although this trivial program doesn't check for errors in the **DosStartSession** call, your program will need to in actual practice because the service is susceptible to a wide variety of errors. For example, OS/2 may not be able to start another session because all its session identifiers may be allocated.

When you terminate the TEST.EXE program by pressing a key, you also terminate the session.

Selecting and Stopping a Session

If your program starts a child session, your program can switch to that session using **DosSelectSession**, whose prototype is

```
unsigned DosSelectSession(unsigned sid,
                          unsigned long reserved);
```

where *sid* is the session identification number of the session to switch to and *reserved* must be 0.

You can use **DosSelectSession** only to switch to a child session or back to the parent. You cannot select an independent session. To switch to the parent, call **DosSelectSession** with *sid* having a value of 0.

The parent session can stop a child session using the **DosStopSession** service, which has the prototype

```
unsigned DosStopSession(unsigned descendants,
                        unsigned sid,
                        unsigned long reserved);
```

If the *descendants* parameter is 0, only the specified session is terminated; if it is 1, the specified session plus any children of that session are

terminated. The *sid* parameter holds the session identification code. The *reserved* parameter must be 0.

To illustrate how **DosSelectSession** and **DosStopSession** work, this program creates a second session and begins running the TEST.EXE program. Next it switches back and forth between the two sessions ten times. Finally the parent session terminates the child, and the program exits.

```
/* This program creates a new session and uses DosSelectSession
   to switch back and forth between the two sessions.
*/

#define INCL_DOS

#include <os2.h>
#include <stdlib.h>

main()
{
  STARTDATA d;
  unsigned sid, pid;
  char flag, ch;

  d.cb = sizeof(d); /* size of struct */
  d.Related = 1;    /* related */
  d.FgBg = 0;       /* foreground */
  d.TraceOpt = 0;   /* no tracing */
  d.PgmTitle = (char far *) "my session"; /* session name */
  d.PgmName = (char far *)"c:\\pm\\test.exe"; /* name */
  d.PgmInputs = (char far *) "";  /* no command line args */
  d.TermQ = (void far *) 0; /* no queue */

  DosStartSession((STARTDATA far *) &d,  /* session data */
                  (unsigned far *) &sid, /* session id */
                  (unsigned far *) &pid);/* process id */

  flag = 0;
  for(ch=0; ch<10; ch++) {
    DosSleep(1000L); /* wait a while */
    flag = !flag;

    if(flag) DosSelectSession(sid, 0L); /* switch to child */
    else DosSelectSession(0, 0L);       /* switch to parent */
  }

  /* return to parent session if not there already */
  DosSelectSession(0, 0L);

  /* kill the child session */
  DosStopSession(0, sid, 0L);
}
```

THREADS

The single most important thing to understand about OS/2's multitasking model is that it is thread (rather than process) based. A thread is

the unit of code dispatched by the scheduler. All the programs you have seen up to this point have consisted of a single thread; that is, the entire program was one thread of execution. This need not always be the case, however, because OS/2 lets the programmer define threads of execution within a program. This allows a single program to create concurrently executing routines, which can, if used correctly, greatly enhance the efficiency of your program. In fact OS/2 also allows you to set the priority of the threads within a program so that you can choose what routines get the greatest access to the CPU. The thread-based services are listed in Table 7-2.

In the first half of this chapter you saw how to create concurrently executing processes. While multitasking processes is a wonderful improvement over single-tasking them and allows a number of divergent applications to share CPU time, it is not generally the approach to take when you want to multitask pieces of a single application. Instead you should use multiple threads within the application.

Another important point about threads and processes is that each process can have up to 255 separate threads, but there can be only about 12 (depending on how your system is configured) separate processes. So when you want to have many paths of execution, use multiple threads rather than multiple processes.

Each thread inherits the environment of the process of which it is a part. This includes open files and environmental strings. If one thread in a process opens a file, for example, other threads can use that file handle. All threads in a program share the same code and data segments, so access to global data and routines is unrestricted.

The thread that begins a process's execution is called either the *main thread* or *thread 1*. It is a little special, as you will soon see.

Table 7-2. OS/2 Thread-Based Services

Service	Function
DosCreateThread	Creates a thread of execution
DosGetPrty	Returns a thread's priority
DosResumeThread	Restarts a suspended thread
DosSetPrty	Sets a thread's priority
DosSuspendThread	Suspends a thread's execution

Creating Threads with DosCreateThread

To create a thread of execution OS/2 uses the **DosCreateThread** service, whose prototype is

```
unsigned DosCreateThread(void far *func(void),
                         unsigned far *tid,
                         char far *stack);
```

where *func* is a pointer to a function that is the entry point into the thread. The function must be declared as **void far** with no parameters. Upon return from the call, *tid* will point to the thread's identifier. The region pointed to by *stack* is used as the thread's stack space. The *stack* parameter points to the top of the stack. Each thread uses its own stack. This region must be at least 512 bytes long, but you really should allow at least 2048 bytes if you will be using any of the API services inside the thread.

The newly created thread begins to execute immediately after it is created. You must not call a thread entry function from another routine.

The following short program uses **DosCreateThread** to create and execute two threads. If you are using Microsoft C version 5.1, you must use this command to compile the program,

```
CL -Lp -Gs thread.c
```

assuming that *thread.c* is the name you give to the program.

```
/* This program uses DosCreateThread to activate two
   concurrently executing threads.

   If you are using Microsoft C 5.1, use this command
   to compile this program:

        CL -Lp -Gs thread.c
*/
#define INCL_SUB
#define INCL_DOS

#include <os2.h>

void far thd1(), far thd2();

char stack1[4096], stack2[4096];
unsigned thd_id1, thd_id2;
```

```
main()
{
  DosCreateThread(thd1,
                  (unsigned far *) &thd_id1,
                  (void far *) &stack1[4095]);

  DosCreateThread(thd2,
                  (unsigned far *) &thd_id2,
                  (void far *) &stack2[4095]);

  VioWrtTTy((char far *) "this is the main thread\n\r", 25, 0);
}

void far thd1()
{
  VioWrtTTy((char far *) "this is thread 1\n\r", 18, 0);
}
void far thd2()
{
  VioWrtTTy((char far *) "this is thread 2\n\r", 18, 0);
}
```

Notice that **DosCreateThread** is called with the last byte of the stack arrays. The 80286 stacks grow from high to low, so it is necessary to pass the last address.

Each thread, including the main program thread, terminates when it reaches the end of the function. However, you can terminate a thread explicitly by calling **DosExit**, whose prototype is

```
void DosExit(unsigned mode, unsigned term_code);
```

If *mode* is 0, only the current thread terminates. If it is 1, the entire process terminates. The value of *term_code* is passed to the calling process.

If the main thread terminates, it terminates the process even if other threads in the process are still active. Keep this in mind when designing your multithread applications.

There are two problems with using **DosCreateThread** directly with high-level languages:

1. It is possible that not all high-level language library functions will be reentrant. If a library function is not reentrant, it cannot be called by two different threads at the same time without causing trouble. Although all the API services are reentrant, language run-time libraries may not be. This is the reason that **VioWrtTTy** was used in the sample program rather than **printf()**. Microsoft's standard C library does not work with multiple threads. (Microsoft does, how-

ever, provide a special multithread library, which will be discussed in a moment.)

2. Because each thread has its own stack, a high-level language that performs run-time stack overflow checking will report false stack overflow errors. Generally you can work around this problem by using a compiler option to turn off run-time stack checking. With the Microsoft compiler use the -Gs compiler directive. However, you do lose the advantage of run-time stack overflow checking.

These two problems can make multithread application tedious to develop. However, all is not lost. Most high-level languages have a special function that creates new threads, and provide special run-time libraries that support multiple threads of execution. In Microsoft C the thread creation function is called **_beginthread()** and has the prototype

```
int cdecl far _beginthread(void far *func(void far *),
                           void far *stack_end,
                           unsigned stack_size,
                           void far *args);
```

The *func* parameter points to the entry function, which must be declared as **void far**. However, the *stack_end* parameter is a pointer to the last byte in the stack, unlike the *stack* parameter in **DosCreateThread**. The *stack_size* parameter must hold the length of the stack in bytes. The *args* parameter points to any information you need to pass to the thread and may be null. The prototype for **_beginthread()** is in PROCESS.H, and you must include this header in any program that uses the function.

Keep in mind that **_beginthread()** does eventually call **DosCreateThread**.

The rest of the examples in this chapter use **_beginthread()** because it is designed to help avoid the problems discussed earlier. If you are using a different compiler, consult your user manual for instructions.

Microsoft C version 5.1 (or greater) supplies a multithread set of library functions as well as a multithread version of all the standard C header files. If you have installed the compiler on your computer in the suggested way, the multithread header files are in the MT\INCLUDE

directory. However, Microsoft C automatically supplies the \INCLUDE, so you need add only the MT\. If your program needs STDIO.H, for example, you will use this **#include** statement:

```
#include <mt\stdio.h>
```

To gain access to the multithread libraries and to reset some compiler options to accommodate multithread applications, use this batch command to compile and link your multithread programs:

```
cl -Alfw %1.c /Link /NOD llibcmt doscalls
```

This command tells the linker to avoid using the default libraries and substitute the LLIBCMT.LIB (the multithread version of the standard C library) and the DOSCALLS.LIB API services library. (Remember, it is possible that DOSCALLS.LIB will be called something else in your version of OS/2.)

The **_beginthread()** function returns the thread's identifier number if it is successful or −1 if it is not.

If you have a different compiler, remember that you must consult your user's manual for specific instructions on alternative multithread libraries and header files.

The code shown here uses **_beginthread()** and the multithread libraries to create a program that parallels the one just shown.

```
/* This program uses _beginthread() to activate two
   concurrently executing threads.
*/
#define INCL_SUB
#define INCL_DOS

#include <mt\os2.h>
#include <mt\process.h>
#include <mt\stdio.h>

void far thd1();
void far thd2();

char stack1[4096], stack2[4096];

main()
{
  _beginthread(thd1,
               (void far *) stack1,
               4096,
               (void far *) 0);
```

```
  _beginthread(thd2,
               (void far *) stack2,
               4096,
               (void far *) 0);

  printf("this is the main thread\n");
}

void far thd1()
{
  printf("this is thread 1\n");
}

void far thd2()
{
  printf("this is thread 2\n");
}
```

Microsoft C also includes a special thread-termination function called **_endthread()**, which has the prototype

void cdecl far _endthread(void);

However, you will probably find **DosExit** more useful since it returns a termination code.

As stated earlier, when the main process thread terminates, all threads in the process terminate. To see an example, run this program:

```
/* This program prints 1000 numbers on the screen using
   a thread.  However, if you press a key, the main thread
   terminates, which stops the entire process.
*/
#define INCL_SUB
#define INCL_DOS

#include <mt\os2.h>
#include <mt\process.h>
#include <mt\stdio.h>

void far thd1(void), far thd2(void);

char stack1[4096], stack2[4096];
unsigned thd_id1, thd_id2;

main()
{
  _beginthread(thd1,
               (void far *) &stack1[4094],
               4096,
               (void far *) 0);
  getch();
}

void far thd1()
{
```

```
  int i;

  for(i=0; i<1000; i++) printf("%d ", i);
}
```

The program begins a thread and then waits for a keypress inside the main thread. If you press a key before **thd1** terminates, the program will terminate. However, if **thd1** finishes, the program will wait until the main thread terminates before exiting.

Unless you specify otherwise, all threads in your program are at the same priority level and are given equal time slices. To illustrate this, watch the output of this program. Both **thd1** and **thd2** print the same number of messages on the screen.

```
/* This program gives you an idea how CPU time is shared
   between two concurrently executing threads.
*/
#define INCL_SUB
#define INCL_DOS

#include <mt\os2.h>
#include <mt\process.h>
#include <mt\stdio.h>
void far thd1(void);
void far thd2(void);

char stack1[4096], stack2[4096];

main()
{
  _beginthread(thd1,
               (void far *) stack1,
               4096,
               (void far *) 0);

  _beginthread(thd2,
               (void far *) stack2,
               4094,
               (void far *) 0);

  getch();
}

void far thd1()
{
  int i;

  for(i=0; i<1000; i++)
    printf("thread 1 - ");
}

void far thd2()
{
  int i;

  for(i=0; i<1000; i++)
    printf("thread 2 - ");
}
```

The comment at the start of the program refers to two threads. The program actually consists of three threads: the main thread, **thd1**, and **thd2**. So why does the comment refer to two threads? The answer is that the main thread is suspended until a key is pressed. Remember that all I/O in OS/2 is interrupt driven. When **getch()** is called, the main thread suspends until you press a key, meaning that CPU time is spent only on the remaining two threads (plus any other processes in the system, of course).

Waiting for Threads to Finish

Since the entire process dies when the main thread dies, it is important to keep the main thread alive until all desired program activity has finished. More generally, it is important for your program to know when the various threads of execution have either completed or are at least in a safe state so that the program can terminate. Although the next chapter covers OS/2 interprocess and interthread communication and synchronization services that provide a solution to this problem, we still need a solution (if only temporarily) for our examples. The one shown here can safely be used in many applications, but should not be construed as a general solution. (The reasons will be made clear in the next chapter.)

The approach and examples developed here have two purposes:

1. They introduce the basic notion of thread synchronization and communication and will make the concept of the *semaphore*, OS/2's standard synchronization method, easier to understand and appreciate.
2. They are excellent illustrations of some key multitasking concepts.

When you need to wait until a thread finishes you generally establish a flag, which the thread sets when it is finished executing. Another thread examines this flag to see whether the other thread is executing. For example, you can rewrite the previous example so that it automatically terminates when both threads have terminated, as shown here:

```
/* This program uses _beginthread() to activate two
   concurrently executing threads and then waits for
   them to end.
*/
#define INCL_SUB
#define INCL_DOS
```

```
#include <mt\os2.h>
#include <mt\process.h>
#include <mt\stdio.h>

void far thd1(void);
void far thd2(void);

char stack1[4096], stack2[4096];

/* These flags will be set to 1 when the two threads terminate */
char term_flag1=0, term_flag2=0;

main()
{
  _beginthread(thd1,
               (void far *) stack1,
               4096,
               (void far *) 0);

  _beginthread(thd2,
               (void far *) stack2,
               4094,
               (void far *) 0);

  printf("this is the main program thread\n");

  while(!term_flag1 || !term_flag2) ; /* wait */
}

void far thd1()
{
  int i;

  for(i=0; i<100; i++)
    printf("thread 1(%d)\n", i);

  term_flag1 = 1;
}

void far thd2()
{
  int i;

  for(i=0; i<100; i++)
    printf("thread 2(%d)\n", i);

  term_flag2 = 1;
}
```

As you can see, the program waits for the other threads to terminate with this wait loop:

```
while(!term_flag1 || !term_flag2) DosSleep(50L); /* wait */
```

However, this leaves much to be desired for two reasons.

1. It keeps the main thread active—and soaking up CPU time—while doing no productive work.
2. Perhaps more important, the **while** loop is computer-bound. Rather than waiting for a keypress, which causes the thread to suspend, the **while** loop keeps the thread constantly ready to run. Remember, a suspended thread demands no CPU cycles. However, a thread that is compute-bound is always able to run and is therefore given CPU cycles. This fact makes the program run much slower than you might think. The next section introduces a solution to this problem.

DosSleep

Throughout this book the **DosSleep** service has been used without much explanation. Now is the time for you to learn how important **DosSleep** can be. The **DosSleep** function causes the thread that calls it to suspend for a specified number of milliseconds. **DosSleep** is not simply a time-delay loop that eats up CPU time; it actually instructs the OS/2 scheduler to suspend the calling thread for the specified time.

To understand how valuable a service **DosSleep** can be, substitute this **while** loop in the previous program and watch how much faster the program runs.

```
while(!term_flag1 || !term_flag2) ; /* wait */
```

Each time through the loop the flags are checked and, if the conditions are not met, the thread sleeps for 50 milliseconds, allowing the other threads greater access to the CPU.

The central issue here is that **DosSleep** is not simply a delay function. Careful use of **DosSleep** allows you to increase the efficiency of your applications. Whenever your program enters a polling loop that is not extremely time critical, you should insert a call to **DosSleep** so that other threads can have more CPU cycles.

Thread Priorities

As you may recall, OS/2 has three categories of execution priorities: idle, regular, and time-critical. Within each category, there are 32 priority levels, 0 through 31. By default all threads within a process have the

same priority: regular, level 0. However, you can alter a thread's priority using the **DosSetPrty** service, which has the prototype

```
unsigned DosSetPrty(unsigned descendants, unsigned class,
                    int p_change, unsigned tid);
```

If the *descendants* parameter is 0, all the threads within the calling process have their priority altered. If *descendants* is 1, all the threads in the calling process plus any child processes are affected. If *descendants* is 2, only the specified thread's priority is changed.

The *class* parameter determines which priority class the specified thread becomes. It can take these values:

Value	Priority Class
0	No change
1	Idle
2	Regular
3	Time-critical

The *p_change* parameter is a signed integer in the range −31 to 31, which will be added to the current priority setting. For example, if *p_change* is 5 and the current priority setting is 7, after the call the new priority will be 12.

The *tid* parameter specifies the process or thread that will have its priority changed.

You can find out a thread's priority using the **DosGetPrty** service, which has the prototype

```
unsigned DosGetPrty(unsigned mode, unsigned far *prty,
                    unsigned tid);
```

If *mode* is 0, the priority of the main thread is returned. If *mode* is 2, the specified thread is returned. The value pointed to by *prty* holds the thread's priority after the call returns. The thread whose priority is desired is specified in *tid*.

The thread's priority is returned with the high-order word holding the general priority class and the low-order word holding the thread's priority within that class. The priority class of a thread is determined by bits 8 and 9 (counting from 0) of the value pointed to by *prty*, as

shown here:

Bit 9	Bit 8	Meaning
0	1	Idle class
1	0	Regular class
1	1	Time-critical class

The following function displays the priority class and level plus the thread's identifier number when passed the priority code returned by **DosGetPrty**. Since the priority class is encoded into the high-order byte of the priority code, when the low-order byte is cleared, the value 256 corresponds to idle, 512 to regular, and 768 to time-critical.

```
void show_priority(unsigned priority, unsigned tid)
{
  unsigned class, level;

  VioSetCurPos(10, 0, 0);
  class = priority & 0xFF00;  /* clear low-order byte */

  printf("Thread %d: Priority class: ", tid);
  switch(class) {
    case 256: printf("Idle\n");
      break;
    case 512: printf("Regular\n");
      break;
    case 768: printf("Time-critical\n");
      break;
  }

  level = priority & 0xFF;  /* clear high-order byte */

  printf("Priority level is: %d\n", level);
}
```

The following program uses the **show_priority()** function to display the priorities of two threads of execution and increases the priority of **thd2**. Although each thread performs the same function—displaying 1000 numbers on the screen—because **thd2** has a higher priority, it completes first because it is given greater access to the CPU.

```
/* This program uses _beginthread() to activate two
   concurrently executing threads.
*/
#define INCL_SUB
#define INCL_DOS
```

```
#include <mt\os2.h>
#include <mt\process.h>
#include <mt\stdio.h>

void far thd1(void);
void far thd2(void);
void clrscr(void);

char stack1[4096], stack2[4096];

/* These flags will be set to 1 when the two threads terminate */
char term_flag1=0, term_flag2=0;
unsigned tid1, tid2;
int thdwait = 1;  /* synchronize the beginning of the threads */

void show_priority(unsigned, unsigned);

main()
{
  unsigned priority1, priority2;

  clrscr();

  tid1 = _beginthread(thd1,
                      (void far *) stack1,
                      4096,
                      (void far *) 0);

  tid2 = _beginthread(thd2,
                      (void far *) stack2,
                      4094,
                      (void far *) 0);

  /* display current priority and class */
  if(DosGetPrty(2, (unsigned far *) &priority1, tid1))
    printf("error getting priority");
  if(DosGetPrty(2, (unsigned far *) &priority2, tid2))
    printf("error getting priority");

  show_priority(priority1, tid1); getch();
  show_priority(priority2, tid2); getch();

  /* now, up the priority of thread 2 by 1 */
  if(DosSetPrty(2, 0, 1, tid2))
    printf("error setting priority");
  if(DosGetPrty(2, (unsigned far *) &priority2, tid2))
    printf("error setting priority");

  show_priority(priority2, tid2); getch();

  /* start the threads */
  thdwait = 0;

  while(!term_flag1 || !term_flag2) DosSleep(10L); /* wait */
}

void far thd1()
{
  int i;

  while(thdwait) DosSleep(10L);
```

```
  for(i=0; i<1000; i++) {
    VioSetCurPos(1, 0, 0);
    printf("thread 1(%d)\n", i);
  }
  term_flag1 = 1;
}

void far thd2()
{
  int i;

  while(thdwait) DosSleep(10L);

  for(i=0; i<1000; i++) {
    VioSetCurPos(1, 60, 0);
    printf("thread 2(%d)\n", i);
  }
  term_flag2 = 1;
}

void show_priority(unsigned priority, unsigned tid)
{
  unsigned class, level;

  VioSetCurPos(10, 0, 0);
  class = priority & 0xFF00;  /* clear low-order byte */

  printf("Thread %d: Priority class: ", tid);
  switch(class) {
    case 256: printf("Idle\n");
      break;
    case 512: printf("Regular\n");
      break;
    case 768: printf("Time-critical\n");
      break;
  }

  level = priority & 0xFF;  /* clear high-order byte */

  printf("Priority level is: %d\n", level);
}

/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

One interesting aspect of this program is that it uses the **thdwait** variable to synchronize the beginning of the two threads. You will learn a better way to synchronize threads in the next chapter.

Suspending Threads

A thread's execution can be suspended by using **DosSuspendThread**, which has the prototype

```
unsigned DosSuspendThread(unsigned tid);
```

where *tid* is the identifier of the thread to be suspended. When a thread is suspended, the scheduler does not grant it access to the CPU. You can suspend only the threads that are within the same process.

A thread suspended by **DosSuspendThread** stays suspended until it is restarted by a call to **DosResumeThread**, which has the prototype

```
unsigned DosResumeThread(unsigned tid);
```

where *tid* is the thread's identifier. **DosResumeThread** can only restart a thread that was previously stopped by a call to **DosSuspendThread**.

To see how these services work, try this program in which **thd1** alternately stops and restarts **thd2** each time through its main loop.

```
/* This program illustrates DosSuspendThread and
   DosResumeThread.
*/
#define INCL_SUB
#define INCL_DOS

#include <mt\os2.h>
#include <mt\process.h> /* included for _beginthread() */
#include <mt\stdio.h>   /* included for printf() */

void far thd1(void);
void far thd2(void);
unsigned thd_id1, thd_id2;

char stack1[4096], stack2[4096];

/* These flags will be set to 1 when the two threads terminate */
char term_flag1=0, term_flag2=0;

main()
{
  thd_id1 = _beginthread(thd1,
                (void far *) stack1,
                4096,
                (void far *) 0);

  thd_id2 = _beginthread(thd2,
                (void far *) stack2,
                4094,
                (void far *) 0);
```

```
  while(!term_flag1 || !term_flag2) ; /* wait */
}

void far thd1()
{
  int i;
  char flag;

  flag = 0;
  DosSleep(1000L);
  for(i=0; i<100; i++) {
    printf("thread 1(%d) - \n", i);
    flag = !flag;
      if(flag) {
        if(DosSuspendThread(thd_id2)) printf("error in suspend");
        else printf("suspending thread 2\n");
      }
      else if(DosResumeThread(thd_id2)) printf("error in restart");
           else printf("restarting thread 2\n");

  }
  term_flag1 = 1;
}

void far thd2()
{
  int i;

  for(i=0; i<30000 && !term_flag1; i++) DosSleep(10L);
  printf("thread 2 reached %d\n", i);
  term_flag2 = 1;
}
```

8

SERIALIZATION AND INTERPROCESS COMMUNICATION

Now that you know the basics of OS/2's multitasking capabilities, it is time to learn about some important concepts and API services that allow you to control multiple-executing processes and threads. As you will see in this chapter, two major issues must be addressed in a multitasking environment:

1. There must be some way to serialize access to certain resources so that only one task has access to the resource at any one time.
2. There must be some way for one process to communicate with another.

The purpose of this chapter is to explore OS/2's solutions to these problems.

THE SERIALIZATION PROBLEM

OS/2 must provide special services that serialize access to a shared resource because, without help from the operating system, a program or thread has no way of knowing that it has sole access to a resource. Imagine writing programs for a multitasking operating system that does *not* provide any serialization support. Imagine further that you have two multiple-executing processes, A and B, both of which require access from time to time to some resource R (such as a disk drive) that must be accessed by only one task at a time. To prevent one program

from accessing R while the other is using it, you try the following solution. First establish a variable called *flag*, which can be accessed by both programs. Your programs initialize *flag* to 0. Then, before a piece of code can access R, it must wait for the flag to be cleared (0) if it is not already cleared. When the flag is cleared, the code sets the flag, accesses R, and when done with R, the program clears the flag. That is, before either program accesses R, it executes this piece of code:

```
while(flag) ; /* wait for flag to be zero */
flag = 1;     /* set flag so another process knows
                 that you are using R
              */
```

The idea behind this code is that neither process accesses R if *flag* is set. Conceptually this approach is in the spirit of the correct solution. In actual fact, however, it leaves much to be desired for one simple reason: It doesn't always work! Let's see why.

Using the code just given, it is possible for both processes to access R at the same time. In essence, the **while** loop performs repeated load and compare instructions on *flag;* in other words, it repeatedly tests *flag*'s value. The next line of code sets *flag*'s value. The trouble is that these two operations could occur in two separate time slices. Between the two time slices, the value of *flag* might have been changed by a different process, thus allowing R to be accessed by both processes at the same time. To understand this, imagine that process A enters the **while** loop and finds that *flag* is 0, which is the green light to access R. However, before it can set *flag* to 1, its time slice expires and process B resumes execution. If B executes its **while** loop, it too finds that *flag* is not set and assumes that it is safe to access R. However, when A resumes it also begins accessing R. The crucial point of the problem is that the testing of *flag* and the setting of *flag* do not comprise one uninterruptible operation. They are two separate operations and, as just illustrated, can be separated by a time slice of the other process. No matter how you try, there is no way, using only application-level code, to guarantee that only one process will access R at a time.

The solution to the serialization problem is as elegant as it is simple: The operating system, in this case OS/2, provides a routine that in one, uninterruptible operation, tests and, if possible, sets a flag. In the language of operating systems engineers, this is called a *test and set* opera-

tion. For historical reasons, the flags used to control serialization are called semaphores. The OS/2 services that allow you to use them are discussed in the next section.

OS/2 SEMAPHORES

OS/2 provides nine services to create and access semaphores. These functions are shown in Table 8-1. The most important use of these services is to allow separate processes or threads to synchronize their activity. As described in the previous section, one important use of semaphores is to control access to a shared resource. They have other uses, however, such as allowing one task to signal another that an event has occurred.

RAM vs. System Semaphores

OS/2 lets you use semaphores to synchronize the action of threads within a process or the action of separate processes. Toward this end, OS/2 supports two different types of semaphores: RAM and system.

The RAM semaphore is used by threads within the same process and is simply a variable of type **unsigned long**. All RAM semaphores

Table 8-1. The OS/2 Semaphore Services

Service	Function
DosCloseSem	Close a system semaphore
DosCreateSem	Create a system semaphore
DosMuxSemWait	Wait for one of several semaphores to be cleared
DosOpenSem	Open a system semaphore
DosSemWait	Wait for a semaphore to be cleared
DosSemRequest	Wait for a semaphore to be cleared, then set it in one uninterruptible operation (Test and Set)
DosSemSet	Set a semaphore
DosSemSetWait	Set a semaphore and wait for it to be cleared
DosSemWait	Wait for a semaphore to be cleared

must be initialized to 0 before they are used. A RAM semaphore in one process has no relationship to a RAM semaphore in another process.

To synchronize activity between two processes, use a system semaphore. To obtain a system semaphore one process must use the **DosCreateSem** service, which returns a handle to the semaphore. The **DosCreateSem** service also initializes the semaphore when it creates it. Other processes access the system semaphore by first calling **DosOpenSem**. To discard a system semaphore, call the **DosCloseSem** service.

Aside from the **DosCreateSem**, **DosOpenSem**, and **DosCloseSem** services, the rest of the semaphore services operate in the same fashion on both RAM and system semaphores.

The main difference between RAM and system semaphores is that RAM semaphores are much faster, so if you only need to synchronize threads within a process, use RAM semaphores.

DosSemSet, DosSemWait, and DosSemClear

One of the first things you learn about the OS/2 semaphore services is that you can't use just one! The semaphore routines work in conjunction with each other so you need to learn about a few of them before you can understand any examples.

To set a semaphore, use the **DosSemSet** service, which has the prototype

```
unsigned DosSemSet(void far *sem);
```

where *sem* is a pointer to the variable that is the semaphore. It is declared as a *void* pointer so that it will work with both RAM and system semaphores without generating compiler errors. Remember that a RAM semaphore is simply an **unsigned long** variable, but a system semaphore is a pointer to a semaphore handle, which in turn is a pointer.

To cause a thread to suspend execution until a specified semaphore is cleared, use **DosSemWait**, whose prototype is

```
unsigned DosSemWait(void far *sem, long timeout);
```

The *sem* parameter must point to the semaphore to wait for. The *timeout* parameter determines how long, in milliseconds, the calling thread suspends if the semaphore is not cleared first. If the value is −1, the service will wait indefinitely.

To clear a semaphore use **DosSemClear**, whose prototype is

```
unsigned DosSemClear(void far *sem);
```

The semaphore pointed to by *sem* will be cleared.

The next few sections show how to use these services to synchronize program activity.

A RAM Semaphore Example

As you learned in the previous chapter, one trouble with multithread programs is that the main thread must stay alive and wait for the other threads in the process to terminate. A temporary solution offered in that chapter was that the main thread looped and waited for flags to be set by the other threads. However, although this solution worked in the specific situation, it should not be generalized. A major problem is that it wastes CPU cycles. A better solution is to use semaphores because, when a thread waits for a semaphore, it suspends until that semaphore is cleared. When the thread is suspended, it does not consume any CPU cycles.

This program uses RAM semaphores to signal the termination of the two threads:

```
/* The main program thread waits for the two RAM semaphores
   to be cleared before terminating.
*/
#define INCL_SUB
#define INCL_DOS

#include <mt\os2.h>
#include <mt\process.h>
#include <mt\stdio.h>

void far thd1(void);
void far thd2(void);

char stack1[4096], stack2[4096];

/* RAM semaphores must be initialized to 0 */
unsigned long sem1=0, sem2=0;
```

```
main()
{
  RESULTCODES waitresult;
  unsigned p;
  int tid1;


  /* set the semaphores */
  if(DosSemSet(&sem1)) {
    printf("cannot set semaphore 1");
    exit(1);
  }
  if(DosSemSet(&sem2)) {
    printf("cannot set semaphore 2");
    exit(1);
  }

  tid1 = _beginthread(thd1,
                      (void far *) stack1,
                      4096,
                      (void far *) 0);

  _beginthread(thd2,
                      (void far *) stack2,
                      4094,
                      (void far *) 0);

  /* wait for the semaphores to be cleared by the threads */
  DosSemWait(&sem1, -1L); /* wait indefinitely */
  DosSemWait(&sem2, -1L); /* wait indefinitely */
}

void far thd1()
{
  int i;

  for(i=0; i<100; i++)
    printf("thread 1(%d)\n", i);

  /* clear the semaphore */
  DosSemClear(&sem1);
}

void far thd2()
{
  int i;

  for(i=0; i<100; i++)
    printf("thread 2(%d)\n", i);

  /* clear the semaphore */
  DosSemClear(&sem2);
}
```

The program establishes two RAM semaphores, **sem1** and **sem2**, and initializes them to 0. The main thread sets the semaphores before creating the child threads. Next, the program creates the child threads and waits for the semaphores to be cleared.

You might find it interesting to try a time-out value, such as 100, to see the effect. The program runs for a short while and then terminates when the time-out limit is reached.

Remember that any thread within the same process can access a RAM semaphore. For example, you can modify **thd2** as shown here. In this version **thd2** waits until **thd1** has finished.

```
void far thd2()
{
  int i;

  /* wait for thd1 to finish */
  DosSemWait(&sem1, -1L);

  for(i=0; i<100; i++)
    printf("thread 2(%d)\n", i);

  /* clear the semaphore */
  DosSemClear(&sem2);
}
```

Using System Semaphores

When you need to synchronize the actions of two processes, you must use a system semaphore. To obtain a system semaphore, call **DosCreateSem**, which has the prototype

```
unsigned DosCreateSem (unsigned exclusive,
                       void far **sem_handle,
                       char far *sem_name);
```

If the *exclusive* parameter is 0, the semaphore being created can be modified only by the process that creates it. If the *exclusive* parameter is 1, any process can set or clear the semaphore. The variable pointed to by *sem_handle* receives a pointer to the system semaphore if the call is successful. The name you give to the semaphore is a string pointed to by *sem_name*. All system semaphores use a filename-like naming convention, which takes the general form

```
\sem\sem_name
```

where *sem_name* is the name of the semaphore. For example,

```
\sem \filelock
```

defines a system semaphore called **filelock**. However, in C, you must use \\ inside a string to generate a single \ because C uses the \ as an escape code. So the previous semaphore name in C string format looks like this:

```
"\ \sem \ \filelock"
```

For a second process to access a system semaphore, it must first open it using **DosOpenSem**, which has the prototype

unsigned DosOpenSem(void far **sem, char far *sem_name);

Here, *sem* is a pointer to the pointer that will receive the address of the system semaphore. The string pointed to by *sem_name* specifies which system semaphore is to be opened.

The following program creates a system semaphore called **handle**, executes a child process called TEST, and waits for the child process to end. The TEST program clears the semaphore just before it terminates.

```
/* This program asynchronously executes another and
   uses a system semaphore to wait until the child
   process ends. */
#define INCL_BASE

#include <os2.h>

main()
{
  char fail[128];
  RESULTCODES result;
  void far *sem;

  /* create a system semaphore */
  if(DosCreateSem(1, /* non-exclusive */
                  (void far **) &sem, /* pointer to system sem */
                  "\\sem\\MySem")) /* semaphore name */
    {
      printf("error creating system semaphore");
      exit(1);
    }

  DosSemSet(sem);

  if(DosExecPgm((char far *) fail, 128,
            1, /* run asynchronous */
            (char far *) "", /* no command line args */
            (char far *) "", /* no environment args */
            (RESULTCODES far *) &result, /* result */
```

```
            "TEST.EXE"))  /* name of program */
      printf("exec failed");

  DosSemWait(sem, -1L); /* wait */
}
```

The TEST program is shown here:

```
#define INCL_DOS

#include <os2.h>

main()
{

  void far *sem;

  if(DosOpenSem((void far **) &sem, "\\sem\\MySem")) {
    printf("TEST cannot open system semaphore");
    exit(1);
  }

  do {
    printf("Inside TEST process\n");
  } while(!kbhit());
  DosSemClear(sem);
}
```

The process that creates a system semaphore is said to own it. When the process that owns a semaphore terminates, the system semaphore is automatically closed. However, your program can explicitly close a system semaphore by using **DosCloseSem**, whose prototype is

```
unsigned DosCloseSem(void far *sem);
```

where *sem* is a pointer to the system semaphore that is to be closed.

SHARING A RESOURCE: AN EXAMPLE

Now that you know how semaphores are maintained, it is time to learn how to use one to serialize access to a shared resource. The example developed in this section illustrates a very common situation found in multitasking programs: One task produces something that a second task consumes. This is often called a *producer-consumer* relationship. The key point to tasks that have this relationship is that the consumer must wait until the producer has finished producing whatever it produces

before the consumer takes it. That is, you do not want the consumer taking a half-created object. This synchronization is achieved through the use of a semaphore.

DosSemRequest

As was discussed at the start of this chapter, one of the key aspects of semaphore use is that some means of testing and setting a semaphore in one uninterruptible operation must be provided. In the examples given so far, this operation was not needed because the semaphores simply signaled the conclusion of some event. However, to use a semaphore to serialize access to a shared resource, the program needs a way to wait until a semaphore is cleared and then set the semaphore in one operation. To accomplish this, OS/2 provides the **DosSemRequest** service, which has the prototype

```
unsigned DosSemRequest(void far *sem, long timeout);
```

where *sem* is a pointer to the semaphore that is being requested and *timeout* is the number of milliseconds to wait for the semaphore. If *timeout* is negative, the service waits indefinitely.

When your program calls **DosSemRequest**, it waits (if necessary) for the specified semaphore to be cleared. When this happens, it then sets the semaphore. At no time will two calls to **DosSemRequest** succeed simultaneously.

It is **DosSemRequest** that enables a program to sequence access to a shared resource. The basic method of operation is to put a call to **DosSemRequest** at the beginning of any code that accesses a shared resource. This way the code executes only when it has control of the resource. At the end of this code put a call to **DosSemClear** to release the semaphore. The code that lies between the call to **DosSemRequest** and **DosSemClear** is often called a *critical section*. The general approach is shown here:

```
Task A                          Task B

DosSemRequest(...)              DosSemRequest(...)
   /* critical section */          /* critical section */
DosSemClear(...)                DosSemClear(...)
```

The Producer-Consumer Program

To illustrate the producer-consumer situation, the following short program creates two threads called, appropriately, **producer** and **consumer**. The producer generates random numbers and stores them in a global variable called **rnd**. The consumer draws horizontal lines on the screen based on the value of **rnd**. The central issue here is that **rnd** is a shared resource and you want to ensure that only one task at a time is accessing it.

The program shown here uses a RAM semaphore called **sem** to control access to **rnd**. Notice that the program source code is indented a level between the calls to **DosSemRequest** and **DosSemClear**. When you use **DosSemRequest** to control access to a resource, you are implicitly defining a block of code, so indentation is a good idea.

```
/* This program creates two threads: a producer and a consumer.
   The producer generates random numbers and the consumer uses
   these numbers to draw lines on the screen.
*/
#define INCL_SUB
#define INCL_DOS

#include <mt\os2.h>
#include <mt\process.h>
#include <mt\stdio.h>
#include <mt\stdlib.h>

void far producer(void);
void far consumer(void);
void clrscr(void);

char stack1[4096], stack2[4096];

/* this is a shared resource */
int rnd;

/* RAM semaphore */
unsigned long sem=0L;

main()
{
   clrscr();

  _beginthread(producer,
               (void far *) stack1,
               4096,
               (void far *) 0);

  _beginthread(consumer,
               (void far *) stack2,
               4094,
               (void far *) 0);
```

```
  while(!kbhit()) DosSleep(500L) ; /* wait */
}

/* This thread produces numbers. */
void far producer()
{
  for(;;) {
    DosSemRequest((unsigned long far *) &sem, -1L);
      rnd = rand();
    DosSemClear((unsigned long  far *) &sem);
  }
}

/* This thread draws lines based upon the numbers produced
   by the producer thread.
*/
void far consumer()
{
  int i;

  for(;;) {
    DosSemRequest((unsigned long far *) &sem, -1L);
      /* clear the previous line */
      VioSetCurPos(10, 0, 0);
      for(i=0; i<80; i++)
      printf(" ");

      /* transform the value in rnd into a number in
         the range 0 through 79 so that the line
         will fit on the screen
      */
      i = rnd % 80;

      /* display the new line */
      VioSetCurPos(10, 0, 0);
      for(; i; i--)
        printf("*");
      DosSleep(1000L); /* just pause a little */
    DosSemClear((unsigned long far *) &sem);
  }

}

/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

You might find it interesting to try this program without using the semaphore.

In this simple example, no serious harm is done if access to the shared resource is not serialized. In almost all real-world applications, however, lack of serialization spells disaster. For example, failure to serialize access to the printer correctly will intermix the output of several tasks.

USING DosEnterCritSec AND DosExitCritSec

OS/2 also provides a second, different method of synchronizing multiple threads within a single process. In this second approach, your program temporarily halts the execution of all but one thread within the process, thus preventing a shared resource from being accessed by two different threads at the same time. The OS/2 services **DosEnterCritSec** and **DosExitCritSec** are used to stop and restart, respectively, all threads in a process except the one that calls these services. Their prototypes are

```
void DosEnterCritSec(void);
void DosExitCritSec(void);
```

Neither service takes a parameter or returns a value.

The best use of these services is when there is a short critical section of code that accesses some shared resource. To ensure that the critical section is safe from interruption, **DosEnterCritSec** is called at the beginning of the code, suspending all other threads. When the critical section has ended, **DosExitCritSec** is called, restarting all other threads. The general approach is

```
DosEnterCritSec();
   /* critical section code is put here */
DosExitCritSec();
```

Keep in mind that **DosEnterCritSec** can be called at several places in your program. Because it suspends the execution of all threads except the caller, there is no chance that a second thread will call **DosEnterCritSec** when the first is in a critical section.

This program shows how **DosEnterCritSec** and **DosExitCritSec** work. Here **thd1** suspends the execution of **thd2** until it has completed.

This effectively serializes the execution of **thd1** and **thd2**, and the advantages of multitasking are lost.

```
/* This program demonstrates the DosEnterCritSec and
   DosExitCritSec services.  Thd1 will complete before Thd2
   because Thd1 halts the execution of the other threads
   in the program.
*/
#define INCL_SUB
#define INCL_DOS

#include <mt\os2.h>
#include <mt\process.h>
#include <mt\stdio.h>

void far thd1(void);
void far thd2(void);

char stack1[4096], stack2[4096];

/* RAM semaphores must be initialized to 0 */
unsigned long sem1=0, sem2=0;

main()
{
  RESULTCODES waitresult;
  unsigned p;
  int tid1;


  /* set the semaphores */
  if(DosSemSet(&sem1)) {
    printf("cannot set semaphore 1");
    exit(1);
  }
  if(DosSemSet(&sem2)) {
    printf("cannot set semaphore 2");
    exit(1);
  }

  tid1 = _beginthread(thd1,
                  (void far *) stack1,
                  4096,
                  (void far *) 0);

  _beginthread(thd2,
                  (void far *) stack2,
                  4094,
                  (void far *) 0);

  /* wait for the semaphores to be cleared by the threads */
  DosSemWait(&sem1, -1L); /* wait indefinitely */
  DosSemWait(&sem2, -1L); /* wait indefinitely */
}

void far thd1()
{
  int i;
```

```
   DosEnterCritSec();
     for(i=0; i<100; i++)
       printf("thread 1(%d)\n", i);
   DosExitCritSec();

   /* clear the semaphore */
   DosSemClear(&sem1);
}

void far thd2()
{
   int i;

   for(i=0; i<100; i++)
     printf("thread 2(%d)\n", i);

   /* clear the semaphore */
   DosSemClear(&sem2);
}
```

A somewhat more interesting example using **DosEnterCritSec** is a modification of the producer-consumer program developed earlier. Instead of using semaphores to control access to the shared variable **rnd**, the modified program shown here uses **DosEnterCritSec** and **DosExitCritSec**.

```
/* This program creates two threads: a producer and a consumer.
   The producer generates random numbers and the consumer uses
   these numbers to draw lines on the screen.

   It uses DosEnterCritSec and DosExitCritSec to serialize
   access to the rnd variable.
*/
#define INCL_SUB
#define INCL_DOS

#include <mt\os2.h>
#include <mt\process.h>
#include <mt\stdio.h>
#include <mt\stdlib.h>

void far producer(void);
void far consumer(void);
void clrscr(void);

char stack1[4096], stack2[4096];

/* this is a shared resource */
int rnd;

main()
{
  clrscr();

_beginthread(producer,
             (void far *) stack1,
```

```
                4096,
                (void far *) 0);

  _beginthread(consumer,
                (void far *) stack2,
                4094,
                (void far *) 0);

  while(!kbhit()) DosSleep(100L); /* wait */
}

/* This thread produces numbers. */
void far producer()
{
  for(;;) rnd = rand();
}

/* This thread draws lines based upon the numbers produced
   by the producer thread.
*/
void far consumer()
{
  int i;

  for(;;) {
    DosEnterCritSec();
      /* clear the previous line */
      VioSetCurPos(10, 0, 0);
      for(i=0; i<80; i++)
        printf(" ");

      /* transform the number into something that will
         fit on the screen
      */
      i = rnd % 80;

      /* display the new line */
      VioSetCurPos(10, 0, 0);
      for(; i; i--)
        printf("*");
    DosExitCritSec();
    DosSleep(100L);
  }
}

/* A simple way to clear the screen by filling
   it with spaces.
*/
void clrscr()
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}
```

You generally want the critical section code to be as short as possible so the rest of the threads do not remain idle for extended periods of time.

Note: For the vast majority of situations, you should use semaphores to synchronize multiple tasks, not **DosEnterCritSec**. The reason for this is quite simple: **DosEnterCritSec** stops all threads in the process whether they need to be stopped or not. This degrades the total performance of your program. The critical section services are in OS/2 for those special situations in which you want to stop the execution of all other threads for a reason, such as a catastrophic error. They should not become your main method of serializing tasks.

INTERPROCESS COMMUNICATION

As you saw earlier in this chapter, system semaphores allow one process to communicate with another process, mostly to achieve some form of synchronized activity. However, OS/2 supports three other forms of interprocess communication: shared memory, pipes, and queues. This section takes a look at shared memory and pipes. OS/2 queues are a more advanced concept and are beyond the scope of this book.

Shared Memory

By default the memory used by one process is logically separate from that used by another. (OS/2 might actually use the same piece of memory for two or more processes because of swapping, but from a logical point of view neither program can actually touch another's memory). However, you can create a shared block of memory that two or more processes can access and use to exchange information. Of all the OS/2 interprocess communication methods, shared memory is the most flexible because it gives you total control of both form and content of the information being shared. However, this freedom comes at a price: Your programs have to handle the data interchanges manually.

To allocate a segment of shared memory, use the **DosAllocShrSeg** service, whose prototype is

```
unsigned DosAllocShrSeg(unsigned size,
                        char far *name,
                        unsigned short selector);
```

The value of *size* specifies the size of the block in bytes. It must be

between 1 and 65,535. The name of the shared segment is specified by *name*, which must take this general form:

\sharemem\seg_name

The variable pointed to by *selector* receives a selector to the allocated segment. For example, this call requests a segment 10 bytes long with the name **MySeg**.

```
DosAllocShrSeg(10, (char far *) "\\sharemem\\MySeg",
               (unsigned short far *) &selector);
```

For a process to obtain a selector to shared memory allocated by another process it must call **DosGetShrSeg**, which has the prototype

```
unsigned DosGetShrSeg(char far *name,
                      unsigned short far *selector);
```

where *name* is the name of the segment and *selector* points to the variable that will receive the segment selector to the shared memory.

Keep in mind that both **DosAllocShrSeg** and **DosGetShrSeg** return a selector to the shared memory. The selector "points" to the first byte of the shared memory segment. (Selectors are discussed in detail in Part One of this book.) Although the selector is sufficient to identify the segment, a selector is not an address as far as C is concerned. To convert the selector into an address you need to use the special **MAKEP** macro. (This may be called something different by your compiler.) The **MAKEP** macro has the prototype

```
void far *MAKEP(unsigned short selector, unsigned offset);
```

where *selector* is a valid memory selector that is combined with *offset* to return a C **far** pointer. For most purposes, *offset* is 0.

The following program allocates a shared memory segment called **MyMem**, writes a string to it, and then executes a child process called SHRTEST, which reads the string from the shared memory and dis-

plays it on the screen.

```
/* This program writes a string into shared memory and
   then executes a child process.  The child
   process reads the string from the shared memory
   and displays it on the screen.
*/
#define INCL_BASE

#include <os2.h>

main()
{
  register int i;
  char fail[128];
  RESULTCODES result;
  unsigned long sem;
  unsigned short shrmem;
  unsigned char far *pshrmem;
  char buf[80], *p;

  /* create a system semaphore */
  if(DosCreateSem(1, /* non-exclusive */
                  (void far **) &sem, /* pointer to system sem */
                  "\\sem\\MySem")) /* semaphore name */
    {
      printf("error creating system semaphore");
      exit(1);
    }

  DosSemSet((void far *) sem);

  if(DosAllocShrSeg(1000,  /* size of segment */
                    "\\sharemem\\MyMem", /* name */
                    (unsigned short far *) &shrmem))
     printf("allocation to shared segment failed\n");

  /* transform the selector into a pointer */
  pshrmem = (char far *) MAKEP(shrmem, 0);

  /* put a string into shared memory */
  strcpy(buf, "this is a test of shared memory");
  p = buf;
  while(*p) *pshrmem++ = *p++;
  *pshrmem = '\0'; /* null terminate the string */

  if(DosExecPgm((char far *) fail, 128,
             1, /* run asynchronous */
             (char far *) "", /* no command line args */
             (char far *) "", /* no environment args */
             (RESULTCODES far *) &result, /* result */
             "SHRTEST.EXE"))  /* name of program */
     printf("exec failed");

  DosSemWait((void far *) sem, -1L); /* Wait  */
}
```

The SHRTEST program is shown here:

```
#define INCL_DOS

#include <os2.h>

main()
{
  unsigned numread;
  unsigned long sem;
  unsigned short shrmem;
  char far *pshrmem;

  if(DosOpenSem((void far **) &sem, "\\sem\\MySem")) {
    printf("SHRTEST cannot open system semaphore");
    exit(1);
  }

  printf("Inside the child process\n");
  printf("\nData read from shared RAM: ");

  if(DosGetShrSeg((char far *) "\\sharemem\\MyMem",
                  (unsigned short far *) &shrmem))
     printf("error obtaining shared memory selector");

  /* transform the selector into a pointer */
  pshrmem = (char far *) MAKEP(shrmem, 0);

  while(*pshrmem) printf("%c", *pshrmem++);

  /* clear the semaphore */
  DosSemClear((void far *) sem);
}
```

Even though these sample programs use shared memory for string data, you can use shared memory to hold any types of objects you desire.

There is one very important thing to remember about using shared memory: You must be sure to allocate enough to hold the largest object you wish to put into it. If your program tries to write past the end of the segment, a memory protection fault is generated, which terminates the process.

Pipes

OS/2 lets two processes communicate with each other via a *pipe*, which is a special type of file maintained by the operating system. Once the pipe has been created, routines read and write to and from the pipe using the standard **DosRead** and **DosWrite** services.

To create a pipe, use **DosMakePipe**, whose prototype is

```
unsigned DosMakePipe(unsigned short far *read_handle,
                     unsigned short far *write_handle,
                     unsigned size);
```

Here, the variable pointed to by *read_handle* receives the read handle for the pipe. The variable pointed to by *write_handle* receives the write handle. The length of the pipe is determined by the value of *size*, which must be in the range 0 through 65,504. If your program tries to write data to a full pipe, the writing process suspends until there is room in the pipe.

Pipes are easy to use for communication, but one little problem must be overcome. The process that creates the pipe must have some method of transferring the read or write handle to the second process. This can be done either by using shared memory or by duplicating the file handles.

A Pipe Example Using Shared Memory The following program creates a pipe and allocates a small segment of shared memory to pass the pipe read handle to the child process called PIPETEST. It then writes a message to the pipe. The child process examines the shared memory to obtain the pipe read handle. It then reads the pipe and displays its contents.

```
/* This program uses a pipe to send information to a
   child process.  This program uses shared memory to
   pass the pipe's handle to the child. */
#define INCL_BASE

#include <os2.h>

main()
{
  char fail[128];
  RESULTCODES result;
  unsigned long sem;
  unsigned short rd, wrt;
  unsigned short child_rd;
  unsigned writtn;
  unsigned short shrmem;
  unsigned short far *pshrmem;

  /* create a system semaphore */
  if(DosCreateSem(1, /* non-exclusive */
```

```
                (void far **) &sem, /* pointer to system sem */
                "\\sem\\MySem")) /* semaphore name */
    {
      printf("error creating system semaphore");
      exit(1);
    }

  DosSemSet((void far *) sem);

  if(DosMakePipe((unsigned short far *) &rd, /* read handle */
                 (unsigned short far *) &wrt,/* write handle */
                 10000))  /* 10,000 bytes long */
    printf("cannot open pipe");

  if(DosAllocShrSeg(2, "\\sharemem\\MyMem",
                    (unsigned short far *) &shrmem))
    printf("allocation to shared segment failed\n");

  /* pass read handle to child via shared memory */

  pshrmem = MAKEP(shrmem, 0);
  *pshrmem = rd;

  if(DosExecPgm((char far *) fail, 128,
            1, /* run asynchronous */
            (char far *) "", /* no command line args */
            (char far *) "", /* no environment args */
            (RESULTCODES far *) &result, /* result */
            "PIPETEST.EXE"))  /* name of program */
     printf("exec failed");

    DosWrite(wrt, (void far *) "shared segment", 14,
            (unsigned far *) &wrttn);

    DosSemWait((void far *) sem, -1L); /* Wait */
}
```

The PIPETEST child process is shown here:

```
#define INCL_DOS

#include <os2.h>

main()
{

  unsigned numread;
  unsigned short rd;
  char buf[80];
  unsigned long sem;
  unsigned short shrmem, far *pshrmem;

  if(DosOpenSem((void far **) &sem, "\\sem\\MySem")) {
    printf("PIPETEST cannot open system semaphore");
    exit(1);
  }
```

```
    printf("Inside the child process\n");
    printf("\nData received from the pipe: ");

    if(DosGetShrSeg((char far *) "\\sharemem\\MyMem",
                    (unsigned short far *) &shrmem))
       printf("error obtaining shared memory selector");

    pshrmem = MAKEP(shrmem, 0);

    rd = *pshrmem;

    /* read the pipe */
    DosRead(rd, (char far *) buf, 14, (unsigned far *) & numread);
    buf[14] = '\0';  /* null terminate the string */

    printf(buf);

    /* clear the semaphore */
    DosSemClear((void far *) sem);
}
```

Using DosDupHandle to Pass a Pipe Handle Another way you can "pass" a pipe handle to another process is to have the process that creates the pipe copy the value of a known handle. This can be accomplished using the **DosDupHandle** service, which has the prototype

```
unsigned DosDupHandle(unsigned short old_handle,
                      unsigned short far *new_handle);
```

All the information associated with the original handle is copied to the new handle. After this operation, the handles are interchangeable: What happens to one will affect the other. The variable pointed to by *new_handle* must either hold a valid file handle or the value FFFFH if you want OS/2 to choose a handle. For our purposes we will supply a valid handle. If the handle pointed to by *new_handle* is currently open, the file it is associated with is closed and the handle is reopened with the new information.

The general approach for using **DosDupHandle** to pass a pipe handle to another process is as follows. The process that creates the pipe duplicates the desired handle onto a known handle. The second process simply assumes that this known handle is associated with the pipe. Admittedly, this process feels a bit shaky because the second process simply assumes something that it cannot verify, but in a tightly controlled situation it can be used with confidence.

This program uses **DosDupHandle** to duplicate a pipe handle:

```
/* This program uses a pipe to send information to
   a child process. */
#define INCL_BASE

#include <os2.h>

main()
{
  char fail[128];
  RESULTCODES result;
  unsigned long sem;
  unsigned short rd, wrt;
  unsigned short child_rd;
  unsigned wrttn;

  /* create a system semaphore */
  if(DosCreateSem(1, /* non-exclusive */
                  (void far **) &sem, /* pointer to system sem */
                  "\\sem\\MySem")) /* semaphore name */
    {
      printf("error creating system semaphore");
      exit(1);
    }

  DosSemSet((void far *) sem);

  if(DosMakePipe((unsigned short far *) &rd,
                 (unsigned short far *) &wrt,
                 10000))
    printf("cannot open pipe");

  child_rd = 4; /* use handle 4 */
  /* dup the rd handle */
  if(DosDupHandle(rd, (unsigned short far *) & child_rd))
    printf("cannot dup handle");

  if(DosExecPgm((char far *) fail, 128,
             1, /* run asynchronous */
             (char far *) "", /* no command line args */
             (char far *) "", /* no environment args */
             (RESULTCODES far *) &result, /* result */
             "PIPETST2.EXE"))  /* name of program */
     printf("exec failed");

    DosWrite(wrt, (void far *) "this is a test", 14,
             (unsigned far *) &wrttn);

    DosSemWait((void far *) sem, -1L); /* Wait */
}
```

The child process PIPETST2 is shown here:

```
#define INCL_DOS

#include <os2.h>
```

```
main()
{

   unsigned rd;
   char buf[80];
   unsigned long sem;

   if(DosOpenSem((void far **) &sem, "\\sem\\MySem")) {
     printf("PIPETST2 cannot open system semaphore");
     exit(1);
   }

   printf("Inside the child process\n");
   printf("\nData Received: ");

   /* read the pipe */
   DosRead(4, (char far *) buf, 14, (unsigned far *) & rd);
   buf[14] = '\0';  /* null terminate the string */
   printf(buf);

   /* clear the semaphore */
   DosSemClear((void far *) sem);
}
```

JUST A SCRATCH ON THE SURFACE

This and the previous chapters have introduced you to the most important and fundamental aspects of OS/2's multitasking capabilities. However, you have only scratched the surface of the multitasking environment provided by OS/2. It is not enough just to know how to use the appropriate OS/2 services to create a multithread or multiprocess application. You must learn to use multitasking effectively. You will want to use multitasking to increase performance of your program and to prevent the user from being idle while the program performs some lengthy task. While it is beyond the scope of this book to discuss the various theories and approaches to writing multitasking applications, you should give much thought to how both data and execution flow through your program, looking for discrete tasks that can be executed concurrently. With practice, this process will become second nature.

9

DEVICE MONITORS

Some of the most desirable programs written for DOS are in a class called *terminate-and-stay resident* (*TSR*) utilities. These types of programs load themselves, initialize any necessary data, and then exit with a call to the DOS TSR function. The program lies dormant in memory until you press a special *hot key*, which activates the program and stops whatever the computer is doing at the time. When the TSR is finished, whatever the computer was doing is resumed. What made (and makes) TSR programs so difficult to write for a DOS environment is that DOS was not designed to accommodate them. As a result several more-or-less undocumented features and aspects of DOS had to be used, which made the programs hard to develop and vulnerable to unforeseen circumstances. The designers of OS/2, however, understood the importance of TSR programs and made provisions for them in OS/2.

Two key concepts distinguish a TSR-type program from a regular application program:

1. The program lies dormant in RAM until needed.
2. You activate the program by pressing a special key.

In OS/2 you can easily realize the first concept by creating a separate task that simply suspends the program's execution until it is needed and using a **VioPopUp** call to request the screen. To handle the second concept, OS/2 allows the creation of a *device monitor*. Through a device monitor a process can examine the input stream of an I/O device such

as the keyboard. If a special value is encountered, the monitor can signal a pop-up application to begin executing.

The term *TSR* is not used in the OS/2 environment. Instead, TSR programs are called *pop-up programs.* Sometimes they may be referred to as part of the larger class of *detachable processes.* It is also not uncommon in OS/2-related literature to see these programs referred to simply as *device monitors,* although this term is somewhat misleading since a device monitor does not need to be associated with a pop-up program. (Pop-ups are sometimes informally called device monitors because the creation of pop-up programs was the most important reason for their inclusion in OS/2. Hence the dual use of the term.) The OS/2 device monitor services are shown in Table 9-1.

This chapter covers the creation of device monitors for both the keyboard and the mouse. Several device monitor programs are developed, including a keystroke translator, a keyboard macro expander, and a pop-up calculator. While the chapter emphasizes the use of a device monitor to provide pop-up applications, keep in mind that device monitors are not limited to this function.

DEVICE MONITOR THEORY OF OPERATION

In its simplest form a device monitor consists of three elements: an input buffer, an output buffer, and a short piece of code that reads from the input buffer and writes to the output buffer. The device monitor is put in between the actual hardware device driver and the application program. Output from the device driver is put into the device

Table 9-1. The Device Monitor Services

Service	Function
DosMonClose	Closes a device monitor
DosMonOpen	Opens a device monitor
DosMonRead	Reads from a device
DosMonReg	Registers (activates) a device monitor
DosMonWrite	Writes to a device

monitor's input buffer. The device monitor reads this information and (in its simplest form) passes the information along to its output buffer. From the device monitor's output buffer, the information is passed to the application program. This situation is depicted in Figure 9-1. It is important to understand that more than one device monitor can be associated with any hardware device. In this case the output of one monitor is passed to the input of the other in a chainlike fashion.

A device monitor is useful because of three special capabilities:

1. A device monitor can examine the input stream from a device and cause special actions to take place when a specific value is encountered. For example, a keyboard monitor can watch for the F9 key, which activates a pop-up application.
2. A device monitor can alter the information received from the device, allowing it to act as a filter, or translator, for certain values.
3. A device monitor can manufacture and transmit output that does not come from the hardware device. For example, you can create a keyboard macro program that generates strings when you press special hot keys. (Such a program might generate "DIR *.*" when the F9 key is pressed.)

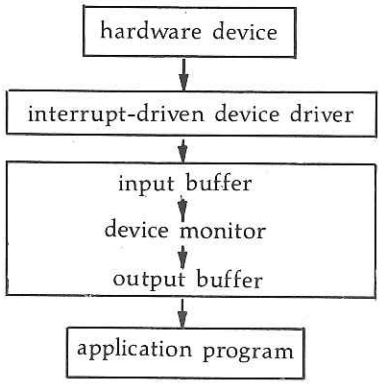


Figure 9-1. How a device monitor becomes part of a device's I/O chain

The creation of a device monitor is essentially a three-step process:

1. You open the monitor to obtain a monitor handle.
2. You register the monitor. The registration process tells OS/2 to enter the monitor into the input chain. In other words, it activates the monitor. After the registration process, the monitor must be ready to begin processing input.
3. The monitor executes a loop that first reads the input buffer and then writes to the output buffer, thus passing along the device information.

In the next sections you will see how to implement this approach.

OPENING AND REGISTERING A DEVICE MONITOR

To open a monitor use **DosMonOpen**, which has the prototype

```
unsigned DosMonOpen((char far *) mon_name,
                    unsigned short far *mon_handle;
```

The *mon_name* parameter must point to a null-terminated string that holds the name of the device to be monitored. The variable pointed to by *mon_handle* receives the monitor's handle upon return from a successful call.

The strings for the devices that can be monitored are shown here:

```
KBD$    MOUSE$

LPT1    LPT2    LPT3    LPT4
```

The most commonly monitored of these are the keyboard (KBD$) and the mouse (MOUSE$).

You can open more than one device monitor for each device. Each monitor is simply placed in the device's input chain.

Before you can use a monitor, you must register it with OS/2 using the **DosMonReg** service, whose prototype is as follows:

```
unsigned DosMonReg(unsigned short mon_handle,
                   void far *inbuf,
                   void far *outbuf,
                   unsigned chain_pos,
                   unsigned sid);
```

The *mon_handle* parameter is the monitor handle returned by a call to **DosMonOpen**. The regions pointed to by *inbuf* and *outbuf* are the monitor's input and output buffers. The size and description of these buffers will be discussed shortly.

OS/2 can insert the monitor into one of three places in the device I/O chain: at the front of the chain, at the end of the chain, or anywhere in between. The value of the *chain_pos* position determines which position is used, as shown here:

chain_pos Value	Position Inserted
0	Anywhere in the chain
1	At the start of the chain
2	At the end of the chain

Keep in mind that even if you specify the start or the end of the chain, a subsequent thread could register another monitor that preempts the earlier one's position.

If you are registering either a keyboard or a mouse monitor, the *sid* parameter must contain the session identifier number of the session to be monitored. It is important to remember that each session establishes its own monitor chain. If you are creating a printer monitor, *sid* will be 1 for the data chain and 2 for the control chain.

Determining the Session Identifier

When you create a keyboard or mouse monitor, you must call **DosMonReg** with the identifier of the session you want to monitor. There are two basic methods of determining the value of the session identifier:

1. You can make use of the fact that the session identifier for the DOS emulator is 2, the first OS/2 session is 4, the second OS/2 session is 5, and so on. The trouble with this method is that it more or less hardcodes the monitor to a specific session.

2. Often you will want to link a monitor to the current foreground session—whatever that session may be. To do this you must first call **DosGetInfoSeg**, which returns selectors to the global and local information segments. One part of the information found in the global information segment is the current foreground session identifier.

DosGetInfoSeg has the prototype

```
unsigned DosGetInfoSeg(unsigned short far *global_seg,
                       unsigned short far *local_seg);
```

After **DosGetInfoSeg** returns, the selectors pointed to by *global_seg* and *local_seg* contain the segment selectors of the global and local information segments.

The information found in the global information segment is arranged like a structure of type **GINFOSEG**, defined by Microsoft. Most of the information it contains is obtainable by other OS/2 services. However, the field **sgCurrent** contains the current session identifier number. You can use this value in a call to **DosMonReg** to associate the monitor with the current session. (Refer to an OS/2 reference for a complete description of the information contained in the global or local information segments.)

Remember that the selector returned by **DosGetInfoSeg** is a selector, not an address as C understands it. You must use the **MAKEP** macro to convert the selector into a pointer. (The **MAKEP** macro is defined in the Microsoft C compiler and may be called something else by different compilers.)

Before going further, let's examine the fragment of code that generates the current session identifier.

```
unsigned short gsel, lsel;
unsigned char sid;

/* get the information segment selectors */
DosGetInfoSeg((unsigned short far *) &gsel,
              (unsigned short far *) &lsel);
info = MAKEP(gsel, 0); /* make a pointer */
sid = info->sgCurrent; /* current session ID */
```

Before you can create a device monitor, you must know the form of the input and output buffers. These are the subjects of the next section.

MONITOR BUFFERS

A device monitor must provide input and output buffers. Each time input is generated, it is put into the input buffer unless that buffer is full. If the buffer is full, the input device is blocked until there is room in the buffer. The next link in the chain reads the output buffer unless it is empty, in which case it waits until there is input. These buffers do not need to be very large for keyboard use because people cannot type faster than the computer can process information. In the examples in this chapter the buffers are only 128 bytes long. However, larger buffers will be desirable for the printer and mouse because these devices can generate data faster than it can be processed.

All monitor buffers must be defined as

```
struct buffers {
  unsigned bufsize;
  char reserved[18];
  char buf[SIZE];
} inbuf, outbuf;
```

Here *bufsize* must hold the size of the entire structure, including itself and the *reserved* parameter. You should define *SIZE* appropriately.

DosMonRead AND DosMonWrite

A device monitor reads input from the device by calling **DosMonRead**, which has the prototype

```
unsigned DosMonRead(void far *inbuf, unsigned wait,
                    void far *packet,
                    unsigned far *length);
```

The region pointed to by *inbuf* must be an input buffer as described in the previous section. If *wait* is 0, **DosMonRead** will wait for data if the queue is empty. If *wait* is 1, the call will return immediately. Generally you will want **DosMonRead** to wait for data. The structure pointed to by *packet* receives the information packet generated by the device. (At no time does your program actually have to examine the input buffer.) The exact nature of the packet will be discussed in the next section. The *length* parameter must point to a variable that contains the length of the

input buffer. On return from the call, *length* contains the number of bytes in the information packet. Even though the packet size does not change for any specific device, a device may generate only part of a packet when it simply wants to signal some event to the device driver.

To pass along input (or output in the case of a printer monitor) you must call **DosMonWrite**, which has the prototype

```
unsigned DosMonWrite(void far *outbuf,
                     void far *packet,
                     unsigned length);
```

The region pointed to by *outbuf* must be the monitor's output buffer. The region pointed to by *packet* must be a valid packet for the device monitored. The value of *length* specifies the length of the packet. You can use the length returned by **DosMonRead** for this value.

DEVICE MONITOR PACKETS

Each device that can be monitored sends information to the monitor in a *packet*. The form of the packet determines what sort of data structure your program will need when using **DosMonRead** and **DosMonWrite**. The different types of packets are discussed here.

The Keyboard Packet

The keyboard packet can be described by the following structure:

```
struct keymonbuf {
  unsigned monflag;   /* device flags */
  char ch;            /* character code */
  char scan;          /* scan code */
  char status;        /* status code */
  char reserved;
  unsigned shift;     /* shift status */
  unsigned long time; /* time of keypress */
  unsigned kbddriver; /* kbd device driver flags */
} ;
```

The meanings of *ch, scan, status, reserved, shift,* and *time* are exactly the same as those returned by the **KbdCharIn** service. (Refer to Chapter 4 for details.)

The *monflag* parameter contains information that is generally applicable only to the device driver interrupt handler.

The *kbddriver* field is encoded as shown here:

Bit	Meaning When Set
0–5	Reserved
6	Key was released
7	Preceding scan code was a prefix
8	Autorepeat generated keystroke
9	Accent key
10–13	Reserved
14–15	User definable bits used for communicating between monitors

For most monitor applications only bit 6 is of interest. As you may already know, the PC keyboard generates two signals for each keypress: a make and a break. The make signal is issued when the key is pressed. The break signal is sent when the key is released. Except for device monitors, you never have to worry about the make and break signals. At the device monitor level, however, your routines often need to know when a key is released, as you will see in the monitor examples developed in this chapter. When bit 6 of *kbddriver* is 0, a key has been pressed. When it is 1, the key has been released.

The Mouse Packet

The mouse information packet can be described by this structure:

```
struct mousebuf {
  unsigned mouflag;
  unsigned event;
  unsigned long time;
  unsigned row;
  unsigned col;
} mybuf;
```

Here, *mouflag* is the device-dependent information used by the device driver. The *time* parameter contains the time of the mouse event, and *row* and *col* contain the mouse's current screen position. The *event* parameter is encoded as shown on the following page.

Bit	Meaning When Set
0	Mouse moved
1	Mouse moved; button 1 pressed
2	No movement; button 1 pressed
3	Mouse moved; button 2 pressed
4	No movement; button 2 pressed
5	Mouse moved; button 3 pressed
6	No movement; button 3 pressed
7–15	Reserved

The Printer Packet

A printer monitor is unique in the sense that it must monitor output rather than input. The printer information packet can be described by the structure

```
struct printer {
        unsigned prnflag;
        unsigned pid;
        char data;
};
```

The *prnflag* parameter contains information specific to the printer device driver. The *pid* field holds the identifier of the process that sent output to the printer. Finally, *data* is the data being transmitted.

DosMonClose

To close a monitor, use the **DosMonClose** service, which has the prototype

```
unsigned DosMonClose(unsigned short mon_handle);
```

where *mon_handle* is the handle of the monitor you wish to close.

A WORD ABOUT EFFICIENCY

Because a device monitor inserts itself into the I/O chain of the monitored device, the performance of the monitor directly affects the effective I/O transfer rate of the device. For this reason a device monitor's code must be very fast. Perhaps even more important, at no time

should a device monitor suspend its operation; doing so effectively breaks the device's I/O chain.

Now that you have learned the necessary background information, it is time to create some device monitors. Most of the examples are keyboard monitors because they represent the most common use of a device monitor.

A First Keyboard Monitor

For an easy first keyboard monitor, let's create one that simply waits for a keystroke. When a keystroke is received, the monitor issues a **VioPopUp**, displays the key and its scan code, waits for a keystroke, and then terminates. This monitor is shown here:

```
/* A very simple keyboard monitor. */

#define INCL_SUB
#define INCL_DOS

#include <os2.h>
#include <stdio.h>

main()
{
  unsigned wait;
  unsigned short mhand;
  struct buffers {
    unsigned size;
    char reserved[18];
    char buf[108];
  } inbuf, outbuf;

  struct keymombuf {
    unsigned monflag;    /* device flags */
    char ch;             /* character code */
    char scan;           /* scan code */
    char status;         /* status code */
    char reserved;
    unsigned shift;      /* shift status */
    unsigned long time; /* time of keypress */
    unsigned kbddriver; /* kbd device driver flag */
  } mybuf;

  unsigned len;
  unsigned short gsel, lsel;
  GINFOSEG far *info;
  unsigned char sid;
  if(DosMonOpen((char far *)"KBD$",
                (unsigned short far *) &mhand))
    {
      printf("cannot open monitor");
      exit(1);
    }
```

```
  inbuf.size = sizeof(struct buffers);
  outbuf.size = sizeof(struct buffers);

  /* get the current screen group ID */
  DosGetInfoSeg((unsigned short far *) &gsel,
                (unsigned short far *) &lsel);
  info = MAKEP(gsel, 0);
  sid = info->sgCurrent;
  if(DosMonReg(mhand,
               (void far *) &inbuf, /* input buffer */
               (void far *) &outbuf,/* output buffer */
               0, /* put the monitor anywhere */
               sid)) /* monitor foreground process */
    {
      printf("cannot register monitor");
      exit(1);
    }

  len = sizeof(struct buffers);

  DosMonRead((void far *) &inbuf,
             0, /* wait for input */
             (void far *) &mybuf,
             (unsigned far *) &len);

  DosMonWrite((void far *) &outbuf,
              (void far *) &mybuf,
              len);

  DosMonClose(mhand);

  wait = 1;  /* non-transparent, wait */

  VioPopUp((unsigned far *) &wait, 0) ;

  printf("You pressed %c\n", mybuf.ch);
  printf("its scan code is %d", mybuf.scan);
  getch();
  VioEndPopUp(0);
}
```

The code is straightforward and should be easy for you to understand. Keep in mind that if you are simply waiting for a keypress it is not technically necessary to follow **DosMonRead** with a call to **DosMonWrite**. However, you must do this if you wish to pass the keystroke along to the next link in the I/O chain. Since only one read-write operation takes place in this example, only the make-keypress signal is detected. However, in the following examples it is necessary to process both the make and break signals.

To try this program, first compile it and then execute it as a detached process. For example, if you call the program MONTEST, exe-

cute it using this command:

```
DETACH MONTEST
```

Next just press a key to activate the pop-up.

A POP-UP APPLICATION SKELETON

The example in the preceding section is very simple in its operation and works fine as it is. However, real device monitors must continue to process device I/O while an application linked to the monitor, such as a pop-up program, executes. As stated earlier, device monitors must consist of very small pieces of efficient code because they are in the I/O chain and any slowing of the I/O system slows the performance of the entire system. You need to do three things to link a monitor to an application:

1. Put the monitor in its own thread with the application in another.
2. Give the monitor thread a higher priority than the application so that it will always have access to the CPU when it needs it to process input on a real-time basis.
3. Have the monitor activate the application by clearing a RAM semaphore. The application must wait for this semaphore to be cleared before it executes and it must reset the semaphore when it finishes.

Using these principles, the following program provides a skeleton you can use to create any type of keyboard monitor pop-up application.

```
/* A keyboard monitor skeleton. */

#define INCL_SUB
#define INCL_DOS

#include <os2.h>

#include <mt\process.h>
#include <mt\stdio.h>

void far keymon();
void far app();
```

```
char stack1[4096], stack2[4096];

unsigned long sem = 0L;
unsigned long term_sem = 0L;
unsigned tid;

main()
{
  DosSemSet((unsigned long far *) &sem);
  DosSemSet((unsigned long far *) &term_sem);

  tid = _beginthread(keymon,
                     (void far *) stack1,
                     4096,
                     (void far *) 0);

  _beginthread(app,
                     (void far *) stack2,
                     4096,
                     (void far *) 0);

  DosSemWait((unsigned far *) &term_sem, -1L);
}

void far keymon()
{
  unsigned char sid;
  unsigned short mhand;
  struct buffers {
    unsigned size;
    char reserved[18];
    char buf[108];
  } inbuf, outbuf;

  struct keymombuf {
    unsigned monflag;    /* device flags */
    char ch;             /* character code */
    char scan;           /* scan code */
    char status;         /* status code */
    char reserved;
    unsigned shift;      /* shift status */
    unsigned long time; /* time of keypress */
    unsigned kbddriver; /* kbd device driver flag */
  } mybuf;

  unsigned len;
  unsigned wait;
  unsigned short gsel, lsel;
  GINFOSEG far *info;


  /* open the monitor */
  if(DosMonOpen((char far *)"KBD$",
                (unsigned short far *) &mhand))
    {
      printf("cannot open keyboard monitor");
      exit(1);
    }

  inbuf.size = sizeof(struct buffers);
  outbuf.size = sizeof(struct buffers);
```

```
    /* get the current screen group ID */
    DosGetInfoSeg((unsigned short far *) &gsel,
                  (unsigned short far *) &lsel);
    info = MAKEP(gsel, 0);
    sid = info->sgCurrent;

    /* increase this thread's priority */
    if(DosSetPrty(2, /* change only this thread */
                  3, /* make time-critical */
                  0, /* leave at lowest priority within class */
                  tid)) /* thread ID */
       printf("could not make keymon() into a time-critical task");

    /* register the monitor */
    if(DosMonReg(mhand,  /* monitor handle */
                 (void far *) &inbuf, /* input buffer */
                 (void far *) &outbuf,/* output buffer */
                 0,     /* put the monitor anywhere */
                 sid)) /* monitor foreground process */
      {
        printf("cannot register keyboard monitor");
        exit(1);
      }

    /* this is the main monitor loop */
    for(;;) {
      len = sizeof(struct buffers);

      mybuf.scan = '\0'; /* clear scan code each time */

      /* read and write the monitor buffers */
      DosMonRead((void far *) &inbuf,
                 0, /* wait for input */
                 (void far *) &mybuf,
                 (unsigned far *) &len);

      DosMonWrite((void far *) &outbuf,
                  (void far *) &mybuf,
                  len);

      /* examine key only after a break code */
      if(!(mybuf.kbddriver & 64)) continue;

      /* In this skeleton, the F10 key deactivates the monitor
         and the F9 key pops up the application.  However,
         you can monitor any keys you like.
      */

      if(mybuf.scan==68) break;  /* exit if F10 is pressed */

      /* if F9 is pressed, let popup application run */
      if(mybuf.scan==67) DosSemClear((unsigned long far *) &sem);
    }

    wait = 1;
    VioPopUp((unsigned far *) &wait, 0);
    printf("closing the keyboard monitor");
    DosMonClose(mhand);
    DosSleep(2000L);
    VioEndPopUp(0);
    DosSemClear((unsigned far *) &term_sem);
}
```

```
/* Popup application */
void far app()
{
  unsigned wait;

  for(;;) {
    /* wait until the monitor receives the hotkey */
    DosSemWait((unsigned long far *) &sem, -1L);

    wait = 1;  /* non-transparent, wait */
    VioPopUp((unsigned far *) &wait, 0) ;

    /* put your popup application code in here */

    printf("strike a key ...");
    getch();

    /* reset the semaphore */
    DosSemSet((unsigned far *) &sem);
    VioEndPopUp(0);
  }
}
```

One of the most important things about this example is the main monitor loop, shown here:

```
/* this is the main monitor loop */
for(;;) {
  len = sizeof(struct buffers);

  mybuf.scan = '\0'; /* clear scan code each time */

  /* read and write the monitor buffers */
  DosMonRead((void far *) &inbuf,
             0, /* wait for input */
             (void far *) &mybuf,
             (unsigned far *) &len);

  DosMonWrite((void far *) &outbuf,
              (void far *) &mybuf,
              len);

  /* examine key only after a break code */
  if(!(mybuf.kbddriver & 64)) continue;

  /* In this skeleton, the F10 key deactivates the monitor
     and the F9 key pops up the application.  However,
     you can monitor any keys you like.
  */

  if(mybuf.scan==68) break;  /* exit if F10 is pressed */

  /* if F9 is pressed, let popup application run */
  if(mybuf.scan==67) DosSemClear((unsigned long far *) &sem);
}
```

Let's look at this loop line by line. First **DosMonRead** must be called with the length parameter set to the length of the buffer. It is reset by **DosMonRead** to return the number of bytes actually read. For this reason, it is necessary to reset the **len** variable before each call to **DosMonRead**. Next come the back-to-back calls to **DosMonRead** and **DosMonWrite**. This is the way input from the keyboard is passed along to the next link in the input chain.

The next line of code is very important. As stated earlier, the keyboard generates both make and break codes. It is important for a pop-up to take place on only one of these. This line of code waits until the break bit in the keyboard device driver variable is set before it allows the key to be examined. This means that the pop-up application activates when the key is released, not when it is pressed. However, you can change this if you like.

Finally, the keystrokes are checked against the predefined hot keys. As it is written, the pop-up application activates when the F9 key is pressed. To terminate the monitor, press the F10 key. You can use any sort of hot key you like. In fact you can have several hot keys and applications linked to one monitor. Just make sure that each application is in its own thread.

One other thing to notice about the skeleton is that the monitor's thread is set to time-critical priority, level 0. This is the lowest-level time-critical setting, and it ensures that the monitor will run before any application task.

A POP-UP CALCULATOR

Using the basic skeleton developed in the previous section, this program monitors the keyboard and pops up a four-function stack-based calculator when the F9 key is pressed:

```
/* A keyboard monitor based popup stack-based calculator
   application. */

#define INCL_SUB
#define INCL_DOS

#define STKMAX 100

#include <os2.h>
```

```
#include <mt\process.h>
#include <mt\stdio.h>
#include <mt\stdlib.h>

void far keymon();
void far app();

char stack1[4096], stack2[4096];

unsigned long sem = 0L;
unsigned long term_sem = 0L;
unsigned tid;

double calcstk[STKMAX];
int tos;

main()
{
  DosSemSet((unsigned long far *) &sem);
  DosSemSet((unsigned long far *) &term_sem);

  tid = _beginthread(keymon,
                     (void far *) stack1,
                     4096,
                     (void far *) 0);

  _beginthread(app,
                     (void far *) stack2,
                     4096,
                     (void far *) 0);

  DosSemWait((unsigned far *) &term_sem, -1L);
}

void far keymon()
{
  unsigned char sid;
  unsigned short mhand;
  struct buffers {
    unsigned size;
    char reserved[18];
    char buf[108];
  } inbuf, outbuf;

  struct keymombuf {
    unsigned monflag;    /* device flags */
    char ch;             /* character code */
    char scan;           /* scan code */
    char status;         /* status code */
    char reserved;
    unsigned shift;      /* shift status */
    unsigned long time; /* time of keypress */
    unsigned kbddriver; /* kbd device driver flag */
  } mybuf;

  unsigned len;
  unsigned wait;
  unsigned short gsel, lsel;
  GINFOSEG far *info;


  /* open the monitor */
  if(DosMonOpen((char far *)"KBD$",
```

```
                (unsigned short far *) &mhand))
    {
      printf("cannot open keyboard monitor");
      exit(1);
    }

  inbuf.size = sizeof(struct buffers);
  outbuf.size = sizeof(struct buffers);

  /* get the current screen group ID */
  DosGetInfoSeg((unsigned short far *) &gsel,
                (unsigned short far *) &lsel);
  info = MAKEP(gsel, 0);
  sid = info->sgCurrent;

  /* up this thread's priority */
  if(DosSetPrty(2, /* change only this thread */
             3, /* make time-critical */
             0, /* leave at lowest priority within class */
             tid)) /* thread ID */
     printf("could not make keymon() into a time-critical task");

  /* register the monitor */
  if(DosMonReg(mhand,
            (void far *) &inbuf, /* input buffer */
            (void far *) &outbuf,/* output buffer */
            0,    /* put the monitor anywhere */
            sid)) /* monitor foreground process */
    {
      printf("cannot register keyboard monitor");
    }

  /* this is the main monitor loop */
  for(;;) {
    len = sizeof(struct buffers);

    mybuf.scan = '\0'; /* clear scan code each time */

    /* read and write the monitor buffers */
    DosMonRead((void far *) &inbuf,
             0, /* wait for input */
             (void far *) &mybuf,
             (unsigned far *) &len);

    DosMonWrite((void far *) &outbuf,
              (void far *) &mybuf,
              len);

    /* wait for a break code */
    if(!(mybuf.kbddriver & 64)) continue;
    if(mybuf.scan==68) break;  /* exit monitor if F10 is pressed */

    /* if F9 is pressed, let popup application run */
    if(mybuf.scan==67) DosSemClear((unsigned long far *) &sem);
  }

  wait = 1;
  VioPopUp((unsigned far *) &wait, 0);
  printf("closing the keyboard monitor");
  DosMonClose(mhand);
  DosSleep(2000L);
  VioEndPopUp(0);
  DosSemClear((unsigned far *) &term_sem);
}
```

```
/* Popup calculator application. */
void far app()
{
  unsigned wait;
  STRINGINBUF strbuf;
  double a, b;
  char str[80];
  char far *endptr;

  strbuf.cb = 80;

  for(;;) {
    DosSemWait((unsigned long far *) &sem, -1L);

    tos = 0;
    wait = 1;  /* non-transparent, wait */
    VioPopUp((unsigned far *) &wait, 0) ;
    VioSetCurPos(2, 0, 0);
    printf("enter 'q' to quit");

    do {
      /* clear entry screen area */
      VioSetCurPos(0, 0, 0);
      printf(":                    ");
      VioSetCurPos(0, 2, 0);
      KbdStringIn((char far *) str,
                  (STRINGINBUF far *) &strbuf,
                  0, 0);

      /* clear answer screen area */
      VioSetCurPos(1, 0, 0);
      printf("                     ");
      switch(*str) {
        case '+':
          VioSetCurPos(1, 0, 0);
          if(!pop(&a) || !pop(&b))
            printf("stack underflow");
          else {
          printf("%lf",a+b);
          push(a+b);
        }
        break;
      case '-':
        VioSetCurPos(1, 0, 0);
        if(!pop(&a) || !pop(&b))
          printf("stack underflow");
        else {
          printf("%lf",b-a);
          push(b-a);
        }
        break;
      case '*':
        VioSetCurPos(1, 0, 0);
        if(!pop(&a) || !pop(&b))
          printf("stack underflow");
        else {
          printf("%lf",b*a);
          push(b*a);
        }
        break;
      case '/':
```

```
            VioSetCurPos(1, 0, 0);
            if(!pop(&a) || !pop(&b))
              printf("stack underflow");
            else {
              if(a==0.0) {
                printf("divide by 0");
                break;
              }
              printf("%lf",b/a);
              push(b/a);
            }
            break;
          case '.': /* display top of stack */
            VioSetCurPos(1, 0, 0);
            if(!pop(&a)) printf("stack underflow");
            else {
              push(a);
              printf("%lf", a);
            }
            break;
          default:
            sscanf(str, "%lf", &a);
            if(!push(a)) printf("stack overflow");
        }
      } while(*str!='q');

      DosSemSet((unsigned far *) &sem);
      VioEndPopUp(0);
    } /* for loop */
}

/* Stack routines for the calculator. */
push(double f)
{

  /* return false if end-of-stack is reached */
  if(tos>=STKMAX) return 0;

  calcstk[tos] = f;
  tos++;
  return 1;
}

pop(double *f)
{
  tos--;

  /* return false if stack underflow occurs */
  if(tos<0) {
    tos = 0;
    return 0;
  }
  *f = calcstk[tos];
}
```

The calculator works with a stack. Each time you enter a number, its value is pushed onto the stack. Each time you enter an operator, the top two values are popped off the stack, the operation is performed,

and the result is displayed. The result is also pushed back onto the stack. For example, to perform the series of additions 10+15+20, you enter the following:

10<enter>
15<enter>
+<enter>
20<enter>
+<enter>

Press . to see what's on the top of the stack. To quit the calculator, press Q. You might find it fun to expand the capabilities of the calculator to accommodate your specific needs. (One good enhancement is a binary-to-hexadecimal convertor.)

To remove the pop-up calculator, press the F10 key.

A SIMPLE KEYBOARD MACRO PROGRAM

One of the most popular utility programs is the keyboard macro program. This type of program associates a string with a special key, such as a function key, and generates that string each time the special key is pressed. For example, you might assign the string "main(int argc, char *argv[])" to the F9 key. Each time you press F9, the string is generated automatically without your having to type it.

The key to creating such a program is to construct keyboard device information packets. By far the easiest way to generate a packet is to use an existing packet, modifying only the fields that you need to change. In the case of a keyboard macro program, you need to change the character code and the make and break bit in the device driver information variable. Keep in mind that when you generate keystrokes, you must send both a make and a break signal.

The simple keyboard monitor shown here inserts the string contained in the global array **mess** into the input stream each time the F9 key is pressed. You can change the contents of **mess** by pressing the F8 key. To terminate the program, press F10.

```
/* A simple keyboard macro program. */

#define INCL_SUB
#define INCL_DOS
```

```
#include <os2.h>

#include <mt\process.h>
#include <mt\stdio.h>

void far keymon();
void far app();

char stack1[4096], stack2[4096];

unsigned long sem = 0L;
unsigned long term_sem = 0L;
unsigned tid;

char mess[80] = "DIR *.*";  /* key macro */
char *str;

main()
{
  DosSemSet((unsigned long far *) &sem);
  DosSemSet((unsigned long far *) &term_sem);

  tid = _beginthread(keymon,
                     (void far *) stack1,
                     4096,
                     (void far *) 0);

  _beginthread(app,
                     (void far *) stack2,
                     4096,
                     (void far *) 0);

  DosSemWait((unsigned far *) &term_sem, -1L);
}

void far keymon()
{
  unsigned char sid;
  unsigned short mhand;
  struct buffers {
    unsigned size;
    char reserved[18];
    char buf[108];
  } inbuf, outbuf;

  struct keymonbuf {
    unsigned monflag;   /* device flags */
    char ch;            /* character code */
    char scan;          /* scan code */
    char status;        /* status code */
    char reserved;
    unsigned shift;     /* shift status */
    unsigned long time; /* time of keypress */
    unsigned kbddriver; /* kbd device driver flag */
  } mybuf;

  unsigned len;
  unsigned wait;
  unsigned short gsel, lsel;
  GINFOSEG far *info;


  /* open the monitor */
```

```
if(DosMonOpen((char far *)"KBD$",
              (unsigned short far *) &mhand))
  {
    printf("cannot open keyboard monitor");
    exit(1);
  }

inbuf.size = sizeof(struct buffers);
outbuf.size = sizeof(struct buffers);

/* get the current screen group ID */
DosGetInfoSeg((unsigned short far *) &gsel,
              (unsigned short far *) &lsel);
info = MAKEP(gsel, 0);
sid = info->sgCurrent;

/* up this thread's priority */
if(DosSetPrty(2, /* change only this thread */
              3, /* make time-critical */
              0, /* leave at lowest priority within class */
              tid)) /* thread ID */
   printf("could not make keymon() into a time-critical task");

/* register the monitor */
if(DosMonReg(mhand,  /* monitor handle */
             (void far *) &inbuf, /* input buffer */
             (void far *) &outbuf,/* output buffer */
             0,     /* put the monitor anywhere */
             sid)) /* monitor foreground process */
  {
    printf("cannot register keyboard monitor");
    exit(1);
  }

/* this is the main monitor loop */
for(;;) {
  len = sizeof(struct buffers);

  mybuf.scan = '\0'; /* clear scan code each time */

  /* read and write the monitor buffers */
  DosMonRead((void far *) &inbuf,
             0, /* wait for input */
             (void far *) &mybuf,
             (unsigned far *) &len);

  if((mybuf.kbddriver & 64) && (mybuf.scan==67)) continue;

  /* if F9 is pressed, insert key macro */
  if(mybuf.scan==67) {
    str = mess;
    for(; *str; str++) {  /* insert the macro string */
      mybuf.kbddriver = mybuf.kbddriver & 191; /* clear break */
      mybuf.ch = *str;
      DosMonWrite((void far *) &outbuf,
                  (void far *) &mybuf,
                  len);
      mybuf.kbddriver = mybuf.kbddriver | 64; /* set break */
      DosMonWrite((void far *) &outbuf,
                  (void far *) &mybuf,
                  len);
    }
  }
```

```
    else
      DosMonWrite((void far *) &outbuf,
              (void far *) &mybuf,
              len);

      /* examine key only after a break code */
      if(!(mybuf.kbddriver & 64)) continue;

      if(mybuf.scan==68) break;  /* exit if F10 is pressed */
      if(mybuf.scan==66) DosSemClear((unsigned long far *) &sem);
    }

    wait = 1; /* wait for screen */
    VioPopUp((unsigned far *) &wait, 0);
    printf("closing the keyboard monitor");
    DosMonClose(mhand);
    DosSleep(2000L);
    VioEndPopUp(0);
    DosSemClear((unsigned far *) &term_sem);
}

/* Change the macro string. */
void far app()
{
    unsigned wait;
    STRINGINBUF strbuf;

    strbuf.cb = 80;
    for(;;) {
      DosSemWait((unsigned long far *) &sem, -1L);

      wait = 1;  /* non-transparent, wait */
      VioPopUp((unsigned far *) &wait, 0) ;
      printf("enter new keyboard macro: ");
      KbdStringIn((char far *) mess,
                (STRINGINBUF far *) &strbuf,
                0, 0);
      mess[strbuf.cchIn] = '\0';
      DosSemSet((unsigned long far *) &sem);
      VioEndPopUp(0);
    }
}
```

The most important bit of code in this example is the part that inserts the string into the input stream. It is shown here:

```
if((mybuf.kbddriver & 64) && (mybuf.scan==67)) continue;

/* if F9 is pressed, insert key macro */
if(mybuf.scan==67) {
  str = mess;
  for(; *str; str++) {  /* insert the macro string */
    mybuf.kbddriver = mybuf.kbddriver & 191; /* clear break */
    mybuf.ch = *str;
    DosMonWrite((void far *) &outbuf,
              (void far *) &mybuf,
              len);
```

```
    mybuf.kbddriver = mybuf.kbddriver | 64; /* generate break */
    DosMonWrite((void far *) &outbuf,
                (void far *) &mybuf,
                len);
  }
}
else
  DosMonWrite((void far *) &outbuf,
              (void far *) &mybuf,
              len);
```

Notice that, unlike the other examples, the code that inserts the string is activated by the pressing—not the releasing—of the F9 key. Once the loop is entered, it generates both make and break signals for each character in the string. In this program, the F9 keystroke is never passed along. The first line of this fragment prevents the break signal from being returned to the input stream. The make signal is not passed along by the code that generates the string, either. You will have to determine whether you want to pass along hot keys or not.

A KEY TRANSLATOR MONITOR

Not all keyboard monitors are used to activate a pop-up application. Some are used to alter the contents of the input stream. The one shown here can perform three different key translations. It can convert all keys into uppercase or lowercase, or it can "encode" each keystroke by adding 1 to the character code of a key. Each translation function is activated and deactivated by a function key, as shown in the comments that begin the program.

```
/* A keyboard monitor that performs various character
   translations.

   Key   Action

   F10   terminate monitor
   F9    turns off lowercasing
   F8    turns on lowercasing
   F7    turns off uppercasing
   F6    turns on uppercasing
   F5    turns off encryption
   F4    turns on encryption

*/
#define INCL_SUB
#define INCL_DOS

#include <os2.h>
```

```
#include <mt\process.h>
#include <mt\stdio.h>

void far keymon();

char stack1[4096];

unsigned long term_sem = 0L;
unsigned tid;

main()
{
  DosSemSet((unsigned long far *) &term_sem);
  tid = _beginthread(keymon,
                     (void far *) stack1,
                     4096,
                     (void far *) 0);

  DosSemWait((unsigned far *) &term_sem, -1L);
}

void far keymon()
{
  unsigned char sid;
  unsigned short mhand;
  struct buffers {
    unsigned size;
    char reserved[18];
    char buf[108];
  } inbuf, outbuf;

  struct keymombuf {
    unsigned monflag;    /* device flags */
    char ch;             /* character code */
    char scan;           /* scan code */
    char status;         /* status code */
    char reserved;
    unsigned shift;      /* shift status */
    unsigned long time; /* time of keypress */
    unsigned kbddriver; /* kbd device driver flag */
  } mybuf;

  unsigned wait;
  unsigned len;
  char lcase;
  char ucase;
  char codeit;

  unsigned short gsel, lsel;
  GINFOSEG far *info;


  /* open the monitor */
  if(DosMonOpen((char far *)"KBD$",
                (unsigned short far *) &mhand))
    {
      printf("cannot open keyboard monitor");
      exit(1);
    }

  inbuf.size = sizeof(struct buffers);
  outbuf.size = sizeof(struct buffers);
```

```
/* get the current screen group ID */
DosGetInfoSeg((unsigned short far *) &gsel,
              (unsigned short far *) &lsel);
info = MAKEP(gsel, 0);
sid = info->sgCurrent;

/* up this thread's priority */
if(DosSetPrty(2, /* change only this thread */
              3, /* make time-critical */
              0, /* leave at lowest priority within class */
              tid)) /* thread ID */
   printf("could not make keymon() into a time-critical task");

/* register the monitor */
if(DosMonReg(mhand,  /* monitor handle */
             (void far *) &inbuf, /* input buffer */
             (void far *) &outbuf,/* output buffer */
             0,     /* put the monitor anywhere */
             sid)) /* monitor foreground process */
  {
    printf("cannot register keyboard monitor");
    exit(1);
  }

lcase = 0;
ucase = 0;
codeit = 0;

/* this is the main monitor loop */
for(;;) {
  len = sizeof(struct buffers);

  mybuf.scan = '\0'; /* clear scan code each time */

  /* read and write the monitor buffers */
  DosMonRead((void far *) &inbuf,
             0, /* wait for input */
             (void far *) &mybuf,
             (unsigned far *) &len);

  if(lcase)
     mybuf.ch = tolower(mybuf.ch);  /* lowercase all letters */
  if(ucase)
    mybuf.ch = toupper(mybuf.ch);   /* uppercase all letters */
  if(codeit)
    mybuf.ch = ++mybuf.ch;          /* code characters */

  DosMonWrite((void far *) &outbuf,
              (void far *) &mybuf,
              len);

  /* examine key only after a break code */
  if(!(mybuf.kbddriver & 64)) continue;

  if(mybuf.scan==68) break;  /* press F10 to exit */

  switch(mybuf.scan) {
    case 67: lcase = 0; /* F9 turns off lowercasing */
      break;
    case 66: lcase = 1; /* F8 turns on lowercasing */
      break;
    case 65: ucase = 0; /* F7 turns off uppercasing */
      break;
```

```
        case 64: ucase = 1; /* F6 turns on uppercasing */
          break;
        case 63: codeit = 0; /* F5 turns off coding */
          break;
        case 62: codeit = 1; /* F4 turns on coding */
      }
    }

    wait = 1;
    VioPopUp((unsigned far *) &wait, 0);
    printf("closing the keyboard monitor");
    DosMonClose(mhand);
    DosSleep(2000L);
    VioEndPopUp(0);
    DosSemClear((unsigned far *) &term_sem);
}
```

A MOUSE DEVICE MONITOR

To conclude this chapter on device monitors, the four-function calculator monitor program is modified to monitor the mouse rather than the keyboard. The converted program is shown here:

```
/* A mouse monitor popup stack-based calculator
   application. */

#define INCL_SUB
#define INCL_DOS

#define STKMAX 100

#include <os2.h>

#include <mt\process.h>
#include <mt\stdio.h>
#include <mt\stdlib.h>

void far keymon();
void far app();

char stack1[4096], stack2[4096];

unsigned long sem = 0L;
unsigned long term_sem = 0L;
unsigned tid;

double calcstk[STKMAX];
int tos;

main()
{
  DosSemSet((unsigned long far *) &sem);
  DosSemSet((unsigned long far *) &term_sem);

  tid = _beginthread(keymon,
                     (void far *) stack1,
                     4096,
                     (void far *) 0);
```

```
  _beginthread(app,
                (void far *) stack2,
                4096,
                (void far *) 0);

  DosSemWait((unsigned far *) &term_sem, -1L);
}

void far keymon()
{
  unsigned char sid;
  unsigned short mhand;
  struct buffers {
    unsigned size;
    char reserved[18];
    char buf[108];
  } inbuf, outbuf;

  struct mousebuf {
    unsigned mou_flag;
    unsigned event;
    unsigned long time;
    unsigned row;
    unsigned col;
  } mybuf;

  unsigned len;
  unsigned wait;
  unsigned short gsel, lsel;
  GINFOSEG far *info;


  /* open the monitor */
  if(DosMonOpen((char far *)"MOUSE$",
                (unsigned short far *) &mhand))
    {
      printf("cannot open mouse monitor");
      exit(1);
    }

  inbuf.size = sizeof(struct buffers);
  outbuf.size = sizeof(struct buffers);

  /* get the current screen group ID */
  DosGetInfoSeg((unsigned short far *) &gsel,
                (unsigned short far *) &lsel);
  info = MAKEP(gsel, 0);
  sid = info->sgCurrent;

  /* up this thread's priority */
  if(DosSetPrty(2, /* change only this thread */
             3, /* make time-critical */
             0, /* leave at lowest priority within class */
             tid)) /* thread ID */
     printf("could not make keymon() into a time-critical task");

  /* register the monitor */
  if(DosMonReg(mhand,
            (void far *) &inbuf, /* input buffer */
            (void far *) &outbuf,/* output buffer */
            0,     /* put the monitor anywhere */
            sid)) /* monitor foreground process */
    {
```

```
    printf("cannot register keyboard monitor");
  }

/* this is the main monitor loop */
for(;;) {
  len = sizeof(struct buffers);

  /* read and write the monitor buffers */
  DosMonRead((void far *) &inbuf,
             0, /* wait for input */
             (void far *) &mybuf,
             (unsigned far *) &len);

  DosMonWrite((void far *) &outbuf,
              (void far *) &mybuf,
              len);

  /* only recognize calculator popup request if mouse
     is in the upper left corner
  */
  if((mybuf.row!=0) || (mybuf.col!=0)) continue;


  if(mybuf.event & 4) break;  /* exit monitor if F1 is pressed */
  if(mybuf.event & 1) printf("mouse moved");
  if(mybuf.event & 16) DosSemClear((unsigned long far *) &sem);
}

wait = 1;
VioPopUp((unsigned far *) &wait, 0);
printf("closing the mouse monitor");
DosMonClose(mhand);
DosSleep(2000L);
VioEndPopUp(0);
DosSemClear((unsigned far *) &term_sem);
}

/* Popup calculator application. */
void far app()
{
  unsigned wait;
  STRINGINBUF strbuf;
  double a, b;
  char str[80];
  char far *endptr;

  strbuf.cb = 80;

  for(;;) {
    DosSemWait((unsigned long far *) &sem, -1L);

    tos = 0;
    wait = 1;  /* non-transparent, wait */
    VioPopUp((unsigned far *) &wait, 0) ;

    VioSetCurPos(2, 0, 0);
    printf("enter 'q' to quit");

    do {
      /* clear entry screen area */
      VioSetCurPos(0, 0, 0);
      printf(":                ");
      VioSetCurPos(0, 2, 0);
```

```
  KbdStringIn((char far *) str,
              (STRINGINBUF far *) &strbuf,
              0, 0);

  /* clear answer screen area */
  VioSetCurPos(1, 0, 0);
  printf("                    ");
  switch(*str) {
    case '+':
      VioSetCurPos(1, 0, 0);
      if(!pop(&a) || !pop(&b))
        printf("stack underflow");
      else {
        printf("%lf",a+b);
        push(a+b);
      }
      break;
    case '-':
      VioSetCurPos(1, 0, 0);
      if(!pop(&a) || !pop(&b))
        printf("stack underflow");
      else {
        printf("%lf",b-a);
        push(b-a);
      }
      break;
    case '*':
      VioSetCurPos(1, 0, 0);
      if(!pop(&a) || !pop(&b))
        printf("stack underflow");
       else {
         printf("%lf",b*a);
         push(b*a);
       }
       break;
     case '/':
       VioSetCurPos(1, 0, 0);
       if(!pop(&a) || !pop(&b))
         printf("stack underflow");
       else {
         if(a==0.0) {
           printf("divide by 0");
           break;
         }
         printf("%lf",b/a);
         push(b/a);
       }
       break;
     case '.': /* display top of stack */
       VioSetCurPos(1, 0, 0);
       if(!pop(&a)) printf("stack underflow");
       else {
         push(a);
         printf("%lf", a);
       }
       break;
     default:
       sscanf(str, "%lf", &a);
       if(!push(a)) printf("stack overflow");
   }
  } while(*str!='q');
```

```
    DosSemSet((unsigned far *) &sem);
    VioEndPopUp(0);
  } /* for loop */
}

/* Stack routines for the calculator. */

push(double f)
{

  /* return false if end-of-stack is reached */
  if(tos>=STKMAX) return 0;

  calcstk[tos] = f;
  tos++;
  return 1;
}

pop(double *f)
{
  tos--;

  /* return false if stack underflow occurs */
  if(tos<0) {
    tos = 0;
    return 0;
  }
  *f = calcstk[tos];
}
```

This monitor requires a mouse application to be running before it will work. If no application is using the mouse, its input is being ignored. Assuming that a mouse application is running and the mouse is in the upper left corner, pressing the right button activates the calculator. Pressing the left button terminates the monitor.

10

CREATING AND USING DYNAMIC LINK LIBRARIES

This chapter examines one of OS/2's most important features: dynamic link libraries. Using dynamic link libraries will make your programs more efficient and more maintainable. The chapter begins with an overview of dynamic linking at both load time and run time and concludes with several examples. It is possible to create dynamic link libraries that have a single thread of execution or multiple threads. However, this chapter is concerned only with single-thread dynamic link libraries.

Throughout the remainder of this chapter, the term *dynlink* is used interchangeably with *dynamic link*. *Dynlink* was coined by the developers of OS/2, and its use seems appropriate.

WHAT IS DYNAMIC LINKING?

Put simply, dynamic linking is the process by which references to external subroutines or data are resolved when the program is loaded. Both static and dynamic linkers have two main functions:

1. They combine separately compiled modules and libraries into an executable program.
2. They resolve references to external functions or data.

For example, suppose you have a main program file that uses library functions. When the program is compiled, only place-holding information is generated when a library function is called because the compiler has no way of knowing where that function will be in memory. It is the linker's job to resolve these addresses.

Dynamic linking is different from static linking in one important way: the time when linking takes place. If a program is statically linked, all functions that it requires are physically bound together in its .EXE file when it is linked. In a dynamic linking situation, however, parts of a program reside in one or more dynlink libraries, which are linked to the main program at load time by the OS/2 loader.

Although final linking is done by the loader, your program still needs to be linked by the linker. When your program calls a dynlink routine, it generates an external reference. When the linker encounters this reference, it generates code that will load the appropriate file when the program is executed. The entire load-time linking process is invisible to the user. To understand just how transparent dynamic linking is, remember that the OS/2 API services are implemented as dynlinks.

DYNLINK ADVANTAGES

Dynamic linking has several advantages over the more traditional static linking. First, there is a great saving in disk space because each program does not contain the code found in the libraries. That is, when several programs that use the same library functions are statically linked, each program file contains copies of the library functions. When the same programs are dynamically linked, there is no duplication of code.

Another important advantage of dynamic linking is that it simplifies the chore of program maintenance. Because the routines in a dynlink library are separate from the main program, you can upgrade or repair a dynlink routine without recompiling the entire program. For example, an accounting package could be upgraded when tax laws change simply by changing a dynlink library. When the program executes, it automatically uses the new routine.

FIVE IMPORTANT FILES

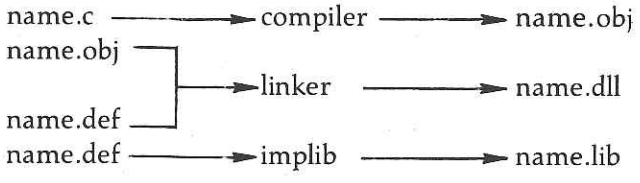
Each dynlink library is supported by a minimum of five separate files, two more than a standard program. First is the file that contains the source code to the dynlink routines. Most likely this will be a C code file. The compiler transforms this file into a standard .OBJ file.

The third file is the definition file associated with the source file. This definition file should have the same name as the source file but use the .DEF extension. The definition file contains several pieces of information that describe the dynlink library. (You will learn more about definition files a little later.)

The fourth file needed by the dynlink library is its *import library*, which is a special type of library file that tells the linker about the dynamic link library functions. This file takes the same name as the source file but ends with .LIB, although it is not a library in the normal sense of the word. You generate this file from the dynlink's definition file by using the IMPLIB utility program supplied by Microsoft.

Finally there is the dynlink library itself. All dynlink library files must use the .DLL extension and must reside in the dynamic link directory. The .DLL file is created by the same linker used to provide static linking. It converts the .OBJ file created by the compiler into a dynlink file.

The creation of the various files is depicted here.



Here *name* is the name of the dynlink library. The actual creation of these files is discussed next.

CREATING A SIMPLE DYNLINK LIBRARY

A simple dynamic link library is developed in this section. Along the way you will learn several important requirements that must be met.

Dynlink Function Declarations

Each dynlink function resides in its own segment, which is separate from the calling program's code segment. Hence all dynamic link functions must be declared as **far**. However, you must also deal with some further complications.

A dynlink function's data is not in the same segment as the calling program. This means that the dynlink function must save the current value of the **DS** register on entry and restore it on exit. To accomplish this, you should put the function type modifier **_loadds**, defined by Microsoft, in front of the function name. For example, this code shows the proper declaration for a dynlink routine called **addit()**:

```
int far _loadds addit(int a, int b);
```

If you are using a compiler other than Microsoft's, study your user manual to see how to declare dynlink functions.

A second thing that you may need to worry about is run-time stack checking. Since the stack for a dynlink function differs from the stack used by the main program, run-time stack checking will generate errors. If you are using the Microsoft compiler, turn off stack checking by using the -Gs compiler option. (Check your user manuals if you are using a different compiler.)

Finally you must instruct the Microsoft compiler to use far pointers and tell it that **SS** does not equal **DS** by using the -Alfw compiler option. (Check your user manuals for instructions if you are using a different compiler.) Keep in mind that the exact compiler options may change with future versions of OS/2.

A Simple Dynlink Library

The following code creates a very small dynlink library that contains only one function: **dllwrite()**. Assume that this file is called DLL.C:

```
#define INCL_SUB
void far _loadds dllwrite(char far *s)
{
  printf(s);
}
```

Before this file can be transformed into a dynlink library, you must create its definition file. Although the next section examines definition files in detail, the one shown here contains the minimum necessary elements to convert DLL.C into a dynlink library.

```
LIBRARY dll
EXPORTS _dllwrite
```

The **LIBRARY** statement specifies the name of the dynlink library. The .DLL extension is assumed. The **EXPORTS** statement lists the functions in the dynlink library that are accessible by other programs. (A dynlink library can contain internal functions that other programs cannot use.) The underscore preceding **dllwrite** is necessary because the Microsoft C compiler (and most others) adds the underscore during compilation. The definition file is case sensitive and must be entered as shown. Call this file DLL.DEF.

To create the dynlink library, use this series of commands:

```
cl -Alfw -Gs -c dll.c
link dll.obj, dll.dll/NOI,,llibcdll.lib doscalls.lib/NOD, dll.def;
```

The linker command line instructs the linker to use DLL.OBJ as input and to create DLL.DLL as output. The /NOI option tells the linker to be case sensitive. The /NOD option causes the default libraries to be ignored and only those specified on the link line are used. Note that different versions of OS/2 and C may call the library DOSCALLS.LIB some other name. The LLIBCDLL.LIB file contains the single-thread dynamic link run-time support library. After this command has executed, you must copy DLL.DLL into the directory specified by the LIBPATH environmental variable found in the CONFIG.SYS file.

Keep in mind that it is not enough just to create the dynlink library. You must also create an import library file to link the main application with the dynlink routines. The IMPLIB utility program generates this file by using the dynlink's definition file. It takes the command line

implib filename.lib filename.def

where *filename* is the name of the dynlink library. Therefore, to create

the import library for DLL.DLL, use this command:

```
implib dll.lib dll.def
```

Accessing Dynlink Functions

Creating the dynlink library and its support files is only half the story. You must follow a few special steps to allow your application program to access the dynlink functions. Each dynlink function used in the program must be declared as an external **far** function. For example, this short program uses the **dllwrite()** function to output a string to the screen. Assume the name of this program is TEST.

```
extern void far dllwrite(char far *);

main()
{
  dllwrite((char far *) "dynlink libraries work");
}
```

Although not required in this situation, it is a good idea to create a definition file for the main program that uses a dynlink library. This file lists the dynlink functions accessed by the program. A valid definition file for this program is shown here:

```
NAME test
IMPORTS dll._dllwrite
```

The first line states the name of the program. The second line specifies which files will be imported from the DLL.DLL dynlink library. (Technically, this line is not needed because the import library created by IMPLIB supplies this information, but a little redundancy for the sake of documentation is not necessarily a bad idea. There are times, however, in which you do need the IMPORTS command.)

When you link the program, you must specify the import library in the library list and include the applications definition file. For example, assuming that the main program is called TEST, use these commands to compile and link it:

```
cl -c test.c
link test.obj/NOI,,,dll.lib slibcep.lib doscalls.lib/NOD,test.def;
```

A Set of Batch Files

To make the creation of applications that use dynlink libraries easier, it is a good idea to create one batch file that compiles and links the library and another that compiles and links the main application. The batch file shown here can be used to create a .DLL library and its import library:

```
cl -Alfw -Gs -c %1.c
link %1.obj, %1.dll/NOI,,llibcdll.lib doscalls.lib/NOD, %1.def;
implib %1.lib %1.def
```

A good name for this batch file is MAKEDLL.CMD.

This batch file compiles the application, specified as the first argument, with the dynlink library named as specified in the second argument:

```
cl -G2 -c %1.c
link %1.obj/NOI,,, %2.lib slibcep.lib doscalls.lib/NOD, %1.def;
```

A good name for this batch file is MAKEMAIN.CMD.

THE DEFINITION FILE

You must create a definition file for each dynlink library. You may also need a definition file for programs. As you have seen, a definition file's most common use is to specify what functions a dynlink library exports or what functions an application file imports. However, several other pieces of information can be included in a definition file.

The linker recognizes 12 definition file commands. Many of the commands are optional. When a command is not included in the definition file, the default setting is used. Let's take a look at these now.

CODE Command

The CODE command tells the linker how to handle the code segments of the associated program or library. It takes the general form

CODE option_list

where *option_list* can be one or more of the following:

Option	Meaning
PRELOAD	The code segment is loaded when the program begins execution (default).
LOADONCALL	The code segment is not loaded until it is called by the program.
SHARED	The code segment can be shared by other programs.
NONSHARED	The code segment cannot be shared by other programs (default).
EXECUTEONLY	The code segment can be executed but not read.
EXECUTEREAD	The code segment can be executed and read (default).
IOPL	The code segment has I/O privileges (not the default).

DATA Command

The DATA command tells the linker how to handle the data segments of the associated program or library. It takes the general form

DATA option_list

where *option_list* may be one or more of the following:

Option	Meaning
PRELOAD	The data segment is loaded when the program begins execution (default).
LOADONCALL	The data segment is not loaded until it is called by the program.
NONE	There is no data segment.
SINGLE	The same data segment is used by all executing versions of the module.
MULTIPLE	Each executing version of the module uses its own data segment.
READONLY	The data segment can be read but not written to.
READWRITE	The data segment can be read and written to.
SHARED	The data segment can be shared by other programs.
NONSHARED	The data segment cannot be shared by other programs (default).
IOPL	The data segment has I/O privileges (not the default).

DESCRIPTION Command

The DESCRIPTION command imbeds the string that follows it in the executable file or library. It takes the general form

```
DESCRIPTION 'string'
```

Notice that the string must be enclosed between single quotes.

The main use for DESCRIPTION is to add copyright information to a program or library prepared for distribution.

EXPORTS Command

The EXPORTS command tells the linker what functions of a module will be accessible by other modules. You can specify up to 3072 exported functions, but each must go on a separate line. The EXPORTS command supports several options, but its simplest form is

```
EXPORTS func_name1
        func_name2
        .
        .
        .
        func_nameN
```

where *func_name* is the name of an exported function.

You can allow a function inside a module to be accessed by a different name by using this form of the EXPORTS command:

```
EXPORTS external_name = internal_name
```

For example, if a function is called **sumit()** inside a library, but your program wants to call it **addit()**, use this EXPORTS statement in the library's definition file.

```
EXPORTS _addit = _sumit
```

The EXPORTS command supports some additional options, but they are for advanced programming situations that are beyond the scope of this book.

HEAPSIZE Command

The HEAPSIZE command determines the number of bytes available for a module's local heap. By default the local heap size is 0. The command takes the general form

```
HEAPSIZE numbytes
```

where *numbytes* is an integer between 0 and 65,536.

Keep in mind that the local heap is separate from the global heap, which is accessed via C's standard dynamic allocation functions.

IMPORTS Command

The IMPORTS command tells the linker what functions the module uses and what files these functions are in. This command is mainly employed when the module calls dynamic link library functions. Its simplest form is

```
IMPORTS filename.func_name
        filename.func_name
        .
        .
        .
        filename.func_name
```

where *filename* is the name of the file that contains the function specified by *func_name*. For example, to import the function **test()** from the library DLL.DLL, use this IMPORT statement:

```
IMPORT DLL._test
```

The underscore is necessary because it is added by the C compiler. The linker automatically adds the .DLL extension to the file name.

You can import any number of functions, but each one must be placed on a separate line and the total number of bytes needed to hold their names must not exceed 65,536.

You can allow a function inside a module to be accessed by a differ-

ent name by using this form of the IMPORTS command:

IMPORTS internal_name = filename.external_name

For example, if a function is called **sumit( )** inside the TEST.DLL library but your program wants to call it **addit( )**, use this IMPORTS statement in the library's definition file:

```
IMPORTS _addit = TEST._sumit
```

The IMPORTS command supports some other options, but their use is beyond the scope of this book.

LIBRARY Command

The LIBRARY command identifies the specified module as a library rather than an application file. It takes the general form

LIBRARY name

where *name* is the name of the library. If no name is specified, the name of the definition file is used.

NAME Command

The NAME command serves two purposes. First, it identifies the associated source file as a program, rather than a library. Second, it can be used to specify the name of the file. The command takes the form

NAME name

where *name* is the name of the application. If no *name* parameter is present, OS/2 uses the name of the executable application file.

PROTMODE Command

If the PROTMODE command is found in a definition file, the associated program can be run only in 80286's protected mode. Otherwise the program may be run in either mode. (To allow this, however, the

program must be processed by the BIND utility and use only the family API services.)

SEGMENTS Command

The SEGMENTS command allows you to define several attributes related to segments. In general you are not likely to need this command. For details refer to an OS/2 technical reference.

STACKSIZE Command

The STACKSIZE command specifies the size, in bytes, of a module's local stack. Generally the size of the local stack is given some value by default, depending on the compiler you are using. You may need to change this, if, for example, a stack overflow error occurs. The general form of the STACKSIZE command is

```
STACKSIZE num_bytes
```

where *num_bytes* must be in the range 0 through 65,536.

STUB Command

The STUB command specifies a DOS-compatible file name that is inserted into an OS/2 application's executable file. If the program is run under DOS, the specified file executes, generally to display a message that the application cannot be run under DOS. The STUB command takes the general form

```
STUB dos_filename
```

where *dos_filename* is the name of a valid DOS-compatible program.

ANOTHER DYNLINK EXAMPLE

For a slightly larger and more meaningful example of creating and using dynlink libraries, several of the screen routines developed in Chapter 3 are put in a dynlink library that can be accessed by any

program you write. The library contains a function to clear the screen, a function to show the current video mode, and one to display the video hardware configuration. The library source code is shown here:

```
/* A dynlink library of video functions */

#define INCL_SUB

#include <os2.h>


/* A simple way to clear the screen by filling
   it with spaces.
*/
void far _loadds clrscr(void)
{
  char space[2];

  space[0] = ' ';
  space[1] = 7;
  VioScrollUp(0, 0, 24, 79, -1, (char far *) space, 0);
}

void far  _loadds showmode(void)
{
  VIOMODEINFO m;

  m.cb = sizeof m;  /* must pass size of struct */
  VioGetMode((VIOMODEINFO far *) &m, 0);
  m.fbType & 1 ? printf("graphics adapter\n"):
                 printf("monochrome adapter\n");
  m.fbType & 2 ? printf("graphics mode\n") :
                 printf("text mode\n");
  m.fbType & 4 ? printf("no color burst\n") :
                 printf("color burst\n");
  printf("%d colors\n", m.color);
  printf("%d columns  %d rows\n", m.col, m.row);
  printf("%d h-res  %d v-res\n\n", m.hres, m.vres);

}

/* Display the video display hardware configuration. */
void far _loadds showconfig(void)
{
  VIOCONFIGINFO c;

  c.cb = sizeof c;

  VioGetConfig(0, (VIOCONFIGINFO far *) &c, 0);

  switch(c.adapter) {
    case 0: printf("Monochrome ");
      break;
    case 1: printf("CGA ");
      break;
    case 2: printf("EGA ");
      break;
```

```
    case 3: printf("VGA ");
      break;
    case 7: printf("8514A ");
  }
  printf("adapter\n");

  switch(c.display) {
    case 0: printf("Monochrome ");
      break;
    case 1: printf("Color ");
      break;
    case 2: printf("Enhanced color ");
      break;
    case 3: printf("PS/2 8503 monochrome ");
      break;
    case 4: printf("PS/2 8513 color ");
      break;
    case 5: printf("PS/2 8514 color ");
  }
  printf("display\n");

  printf("%lu bytes of memory on video adapter\n", c.cbMemory);
}
```

Compile this file by using the MAKEDLL batch file shown earlier in this chapter. (A good name for this library is SCRN.) To link the file you need to create its definition file, as shown here, and run it through IMPLIB.

```
LIBRARY SCRN
EXPORTS _clrscr
        _showconfig
        _showmode
```

Use this short program to try the library:

```
extern void far clrscr(void);
extern void far showmode(void);
extern void far showconfig(void);
main()
{
  clrscr();
  showmode();
  showconfig();
}
```

Create this definition file for the program:

```
NAME scrntest
IMPORTS scrn._clrscr
        scrn._showmode
        scrn._showconfig
```

Use the batch file MAKEMAIN shown earlier to compile and link the program.

RUN-TIME DYNAMIC LINKING

As flexible as load-time dynamic linking is, it is not the answer for all situations because you need to know the name of the module and the name of the functions within the module at compile time. However, some applications need to be able to access a dynlink routine that is defined at run time. For example, a problem-solving AI-based program may access a collection of problem-solving routines in its attempt to find a solution to a given problem. Using run-time dynamic linking, the problem solver could try an arbitrarily long list of different problem-solving functions—even new ones added while it is running—in its attempt to find a solution. In general, run-time dynamic linking allows your program to handle changing situations easily.

To enable run-time dynamic linking OS/2 provides the five services shown in Table 10-1. This section presents these services and develops two examples.

DosLoadModule and DosGetProcAddr

Before your program can access a function that is loaded dynamically at run time, you must load the module containing the function into

Table 10-1. The Run-Time Dynamic Linking Services

Service	Function
DosFreeModule	Disposes of a dynlink module and frees the memory used by it
DosGetModHandle	Returns a dynlink module handle
DosGetModName	Returns the name of the module given its handle
DosGetProcAddr	Returns the address of a specific function within a dynlink module
DosLoadModule	Loads the specified dynlink library

memory by using the **DosLoadModule** service, which has the prototype

```
unsigned DosLoadModule(char far *failbuf,
                       unsigned failbuf_size,
                       char far *name,
                       unsigned far *mhandle);
```

The region of memory pointed to by *failbuf* receives information about the cause of a failure if an error prevents **DosLoadModule** from finishing its load operation. The size of the buffer is specified by *failbuf_size*. Generally 128 bytes is sufficient. The file name of the dynlink library, including drive and path information, must be pointed to by *name*. If successful, **DosLoadModule** returns a module handle to the variable pointed to by *mhandle*.

If the module has already been loaded by another program, it is not reloaded.

Once you have loaded the module, you must use **DosGetProcAddr** to obtain the address of each function in the library you want to call. **DosGetProcAddr** has the prototype

```
unsigned DosGetProcAddr(unsigned mhandle,
                        char far *func_name,
                        type far **(func_addr)());
```

The *mhandle* parameter must be acquired through a call to **DosLoadModule**. The string pointed to by *func_name* contains the name of the function that you want to call. The pointer to that function is the function pointer pointed to by *func_addr*. You must substitute the correct return type of the function for the word *type* shown in the prototype.

To see a simple example of run-time dynamic linking, try this program:

```
/* This program assumes that the dynlink library DLL.DLL,
   developed earlier in this chapter, is available.
   If it is not, you must create it before attempting to
   use this program.
*/

#define INCL_DOS

#include <os2.h>
```

```
char failbuf[128];
unsigned mhandle;

void (far *func) (char far *);

main()
{
  getch();
  if(DosLoadModule((char far *) failbuf, /* name of fail buffer */
                   sizeof(failbuf),      /* size of fail buffer */
                   (char far *) "dll",   /* name of dynlink lib */
                   (unsigned far *) &mhandle)) /* module handle */
  {
    printf("error loading dynlink module");
    exit(1);
  }

  if(DosGetProcAddr(mhandle, (char far *) "_dllwrite",
                    (void far **)  &func))
  {
    printf("cannot find the specified function");
    exit(1);
  }

  func((char far *) "runtime dynlink module loading works");
}
```

As the comment at the start of the program suggests, this program dynamically loads the DLL.DLL dynlink library developed in the first part of this chapter and uses **dllwrite()** from that library to display a message. You should pay special attention to the declaration of the function pointer **func**. Remember that *func* is the name of a pointer to a function, not the name of a function.

DosFreeModule

In the preceding example, the program terminated immediately after calling the dynlink function. In a real application this will probably not be the case. Since a program may need to load several different modules at different times, OS/2 provides the **DosFreeModule** service, which removes a module and frees the memory it used for other modules. The prototype for **DosFreeModule** is

```
unsigned DosFreeModule(unsigned mhandle);
```

where *mhandle* is the handle of the module that is being removed.

Another Run-Time Dynamic Link Example

To help give you a feel for using run-time dynlink libraries, a short file utility dynlink library is created here along with a program that uses it. The file utility library is a collection of functions developed in Chapter 6 that allows you to list the directory, display the contents of a file, and report information about the disk system. Although this example could have been written without using run-time dynamic linking, it does illustrate its use. (Programs that actually need run-time dynamic linking tend to be quite long and complex so they are unsuitable for examples.)

The file dynlink functions are shown here:

```
/* File utility functions. */

#define INCL_DOS

#include <os2.h>

void far _loadds show_dir(void);
void far _loadds displayfile(char *fname);
void far _loadds browse(char *fname);
void far _loadds diskinfo(void);

/* This function displays an entire file. */
void far _loadds displayfile(char *fname)
{
  unsigned short fh;
  unsigned action;
  unsigned num_bytes;
  char buf[513];

  /* open the file, no file sharing */
  if(DosOpen((char far *) fname, /* filename */
          (unsigned short far *) &fh, /* pointer to handle */
          (unsigned far *) &action,   /* pointer to result */
          0L,     /* 0 length */
          0,      /* normal file */
          0x1,  /* open */
          0x10,   /* read-only, no-share */
          0L))    /* reserved */
  {
    printf("error in opening file");
    exit(1);
  }

  do {
    if(DosRead(fh, (char far *) buf, 512,
              (unsigned far *) &num_bytes)) {
      printf("error reading file");
      exit(1);
    }
    buf[num_bytes] = '\0'; /* null terminate the buffer */
```

```
    printf(buf);
  } while(num_bytes);

  if(DosClose(fh)) printf("error closing file");
}


/* A file browse function. */
void far _loadds browse(char *fname)
{
  unsigned short fh;
  unsigned action;
  unsigned num_bytes;
  long pos, p;
  char buf[513], ch;

  /* open the file, no file sharing */
  if(DosOpen((char far *) fname, /* filename */
          (unsigned short far *) &fh, /* pointer to handle */
          (unsigned far *) &action,   /* pointer to result */
          0L,     /* 0 length */
          0,      /* normal file */
          0x1,  /* open */
          0x10,  /* read-only, no-share */
          0L))   /* reserved */
  {
    printf("error in opening file");
    exit(1);
  }

  /* main loop */
  pos = 0L;
  do {
    if(DosRead(fh, (char far *) buf, 512,
               (unsigned far *) &num_bytes)) {
      printf("error reading file");
      exit(1);
    }
    buf[num_bytes] = '\0'; /* null terminate the buffer */
    printf(buf); /* display the buffer */

    /* see what to do next */
    ch = tolower(getch());
    switch(ch) {
      case 'e': /* move to end */
        DosChgFilePtr(fh, -512L, 2, (unsigned long far *) &pos);
        break;
      case 's': /* move to start */
        DosChgFilePtr(fh, 0L, 0, (unsigned long far *) &pos);
        break;
      case 'f': /* move forward */
        /* forward is automatic, so no change is required */
        pos = pos + num_bytes;
        break;
      case 'b': /* move backward */
          pos = pos - 512;
          if(pos<0L) pos = 0L;
          DosChgFilePtr(fh, pos, 0, (unsigned long far *) &p);
      }

    } while(ch != 'q');
```

```
  if(DosClose(fh)) printf("error closing file");
}


/* This routine lists the directory. */
void far _loadds show_dir()
{
  FILEFINDBUF f;
  unsigned short hdir;
  unsigned count;

  printf("\n");
  hdir = 0xffff; /* cause a new handle to be returned */
  count = 1;  /* find the first match */
  DosFindFirst((char far *) "*.*", (unsigned short far *) &hdir,
               0x0, (FILEFINDBUF far *) &f, sizeof(f),
               (unsigned far *) &count, 0L);
  do {
    printf("%-13s %d\n", f.achName, f.cbFile);
    DosFindNext(hdir, (FILEFINDBUF far *) &f, sizeof(f),
                (unsigned far *) &count);
  }while(count);
  DosFindClose(hdir);
}


/* This function displays the number of bytes
   per sector, sectors per cluster, total disk space,
   and available disk space.
*/
void far _loadds diskinfo(void)
{
  FSALLOCATE f;

  DosQFSInfo(0, 1, (char far *) &f,
             sizeof f);

  printf("Bytes per sector: %ld\n", f.cbSector);
  printf("Sectors per cluster: %ld\n", f.cSectorUnit);
  printf("Total disk space: %ld\n",
         f.cbSector * f.cSectorUnit * f.cUnit);
  printf("Total available disk space: %ld\n",
         f.cbSector * f.cSectorUnit * f.cUnitAvail);
}
```

Call this file FILE.C. The definition file for the library is shown here:

```
LIBRARY FILE
EXPORTS _show_dir
        _displayfile
        _browse
        _diskinfo
```

Compile and link FILE.C by using the MAKEDLL batch file.

The program shown here loads FILE.DLL during run time, displays a menu, and calls from the menu the function chosen by the user. Notice that the function pointer *func* does not have a prototype parameter list declared. Since the file functions do not all take the same number of parameters, it is not possible to use a prototype.

```
/* A simple menu driven file manager program that uses
   a runtime dynlink library.
*/

#define INCL_DOS

#include <os2.h>

char failbuf[128];
unsigned mhandle;

void (far *func) ();

main()
{
  char choice;
  char fname[80];

  if(DosLoadModule((char far *) failbuf, /* name of fail buffer */
                sizeof(failbuf),         /* size of fail buffer */
                (char far *) "file",     /* name of dynlink lib */
                (unsigned far *) &mhandle)) /* module handle */
  {
    printf("error loading dynlink module");
    exit(1);
  }

  do {
    choice = menu();
    switch(choice) {
      case 1:
        if(DosGetProcAddr(mhandle, (char far *) "_displayfile",
                          (void far **)  &func))
        {
          printf("cannot find the specified function");
          exit(1);
        }
      printf("\nfilename: ");
      gets(fname);
      func((char far *) fname);
      break;
    case 2:
      if(DosGetProcAddr(mhandle, (char far *) "_browse",
                        (void far **)  &func))
      {
        printf("cannot find the specified function");
        exit(1);
      }
      printf("\nfilename: ");
      gets(fname);
      func((char far *) fname);
      break;
```

```
      case 3:
        if(DosGetProcAddr(mhandle, (char far *) "_show_dir",
                          (void far **)  &func))
        {
          printf("cannot find the specified function");
          exit(1);
        }
        func();
        break;
      case 4:
        if(DosGetProcAddr(mhandle, (char far *) "_diskinfo",
                          (void far **)  &func))
        {
          printf("cannot find the specified function");
          exit(1);
        }
        func();
    }
  } while(choice!=5);
  DosFreeModule(mhandle);
}

/* Display a menu. */
menu()
{
  char choice;

  printf("1. list a file\n");
  printf("2. browse through a file\n");
  printf("3. directory\n");
  printf("4. disk info\n");
  printf("5. quit\n");

  do {
    printf("Enter your selection: ");
    choice = getche();
    printf("\n");
  } while (choice < '1' || choice > '5');
  return choice - '0';
}
```

DYNAMIC LINKING IMPLICATIONS

The use of dynlink libraries in either their load-time or run-time form not only expands the options available to you when you create an application but also implies a fundamental restructuring to the approach of program design. To take the best advantage of dynlinks you need to group the various functional elements of your program into separate dynlink libraries. While this step is fairly obvious, the next is not. You must decide what parts of your program are more-or-less fixed and

what parts may change. Although it is conceivable to have the main program consist simply of a **main()** function that issues calls to dynlink routines, a more likely situation involves a balance between dynlink code and statically linked program code. The proper mix will vary between applications, and achieving it requires both thought and experimentation. Remember that the flexibility and improved maintainability of your programs is worth the extra effort that dynamic linking requires.

III

PROGRAMMING PRESENTATION MANAGER

This final part of the book introduces the Presentation Manager. The Presentation Manager is important for several reasons:

- It provides a windowed environment that can be used quite effectively.
- It provides a dynamic data interchange facility that allows one application to transfer data to another easily.
- It provides graphics services that allow your programs to draw points, lines, boxes, and circles.

The Presentation Manager is a very complex piece of software containing several hundred API services. While it is beyond the scope of this book to cover it in depth, the next two chapters present an overview. The main focus of this section is the basic methodology used to create Presentation Manager-compatible programs. If you will be creating many programs that make use of the Presentation Manager, however, you will find the book *Presentation Manager Programming* (by Herb Schildt, Osborne/McGraw-Hill, 1989) helpful because it provides a thorough examination of all the important Presentation Manager features.

11

PRESENTATION MANAGER: AN OVERVIEW

Beginning with version 1.1, OS/2 includes the Presentation Manager as the default user interface. The Presentation Manager provides the user with a windowed, graphical interface that displays the functionality of the system on the screen and makes the operation of the computer by the user much more intuitive than the traditional command line interface. As you will see, however, the end user's ease of operation has a price—at times a fairly high price—which is paid by the programmer in the extra time and effort it takes to create a Presentation Manager-compatible program. This chapter introduces the fundamental concepts implemented by the Presentation Manager and develops a Presentation Manager application skeleton that you can use to create your own programs.

WHAT IS THE PRESENTATION MANAGER?

What the Presentation Manager is depends to some extent on whether you are an end user or a programmer. From the user's point of view, the Presentation Manager is a shell to interact with in using applications. From the programmer's point of view, however, the Presentation Manager is a collection of several hundred additional API services, coupled with a general application design philosophy. From the programmer's point of view, the Presentation Manager is a giant toolbox of interrelated functions that, when used correctly, allow the creation of application programs that share a common interface.

The goal of the Presentation Manager is to enable a person who has basic familiarity with the system to sit down and run virtually any

application without prior training. In theory if you can run one Presentation Manager program, you can run them all. Of course, in reality most useful programs still require some sort of user instruction, but at least this instruction can be restricted to what the program does, not how the user must interact with it.

At this point it is very important for you to understand that not every Presentation Manager program necessarily presents the user with a Presentation Manager-style interface. You can override the basic Presentation Manager philosophy, but you had better have a very good reason, because the users of your programs will be very disturbed by the change. If you are writing application programs for OS/2, they should conform to the general Presentation Manager application interface philosophy to be successful in the marketplace.

Let's look at a few of the more important features of the Presentation Manager and the design philosophy behind them.

The Desktop Model

With few exceptions, the point of a window-based user interface is to provide on the screen the equivalent of a desktop. On a desk you often find several different pieces of paper, one on top of another, often with fragments of different pages visible beneath the top page. The equivalent of the desktop in the Presentation Manager is the screen. The equivalents of the pieces of paper are windows on the screen. You can move pieces of paper about on a desk, maybe switching which piece of paper is on top or how much of another is exposed to view. The Presentation Manager allows the same types of operations on its windows. By selecting a window you can make it current, which means putting it on top of all other windows. You can also enlarge or shrink a window or move it about on the screen. In short, the Presentation Manager lets you control the surface of the screen the way you control the surface of your desk.

The Mouse

Unlike DOS and the original version of OS/2, the Presentation Manager allows you to use the mouse for almost all control, selection, and

drawing operations. Of course, to say that it allows the use of the mouse is an understatement. The fact is that the Presentation Manager interface was designed for mouse input; it *allows* the use of the keyboard! Although it is certainly possible for an application program to ignore the mouse, it does so only in violation of a basic Presentation Manager design principle.

To activate a feature you generally move the mouse pointer to that feature and double click the left mouse button. A *double click* is achieved by pressing the button twice in rapid succession. The Presentation Manager allows you to *drag* objects about by moving the mouse pointer to the object, pressing and holding the left button, and moving the mouse pointer and object to a new location.

Icons and Graphics Images

The Presentation Manager allows (but does not require) the use of icons and bit-mapped graphics images to convey information to the user. The theory behind the use of icons and graphics images is found in the adage, "a picture is worth a thousand words."

In OS/2 terminology, an *icon* is a small symbol representing some function or program that can be activated by moving the mouse to the icon and double clicking on it. A graphics image is generally used simply to convey information quickly to the user.

Menus and Dialog Boxes

Aside from standard windows, the Presentation Manager also provides special purpose windows. The most common of these are the menu and dialog boxes. Briefly, a *menu* is a special window that contains options from which the user makes a selection. Instead of providing the menu selection functions in your program, you simply create a standard menu window by using Presentation Manager services.

A *dialog box* is essentially a special window that allows more complex interaction with the application than a menu allows. For example, your application might use a dialog box to input a file name. With few exceptions, nonmenu input is accomplished in the Presentation Manager via a dialog box.

STORMY Cs

Now the bad news. Because the Presentation Manager must control all input and output, many of the C standard library functions, such as **printf( )** and **scanf( )** are not usable by any program that is going to run under the Presentation Manager. In fact one reason that there are so many Presentation Manager services is that they must replace a large number of the standard C functions.

GENERAL OPERATION OF A PRESENTATION MANAGER APPLICATION

You must fix firmly in your mind one important point: The flow of a Presentation Manager application program is fundamentally different from a "normal" application. You need to abandon your preconceived notions about how information moves in and out of your program and what constitutes a program's "main loop." Before looking at any concrete Presentation Manager services or examples, you must understand the structure of all Presentation Manager-compatible programs.

An Overview of the Operation of a Presentation Manager Application

All programs that are compatible with the Presentation Manager share a common skeleton. In its most straightforward implementation, when the compatible program begins, it performs the following functions in the order shown:

1. Initializes the Presentation Manager relative to the program
2. Establishes a message queue
3. Registers a special function called the *window function* within the application called by the Presentation Manager
4. Creates a window of the registered class
5. Executes a loop that reads messages from the queue and dispatches them to the window function

The window function (sometimes called *wind-proc* or *windowproc*) is a special function that is called only by the Presentation Manager, not by your program. It receives in its parameters a message from the message queue established in the second step. It then takes different actions based on the value of each message. The form and content of these messages, as well as the window function, will be discussed shortly.

When a Presentation Manager application ends, it must perform the following three steps:

1. Destroy the window
2. Destroy the message queue
3. Terminate the window environment relative to the application

The Message Loop

Aside from creating and destroying the windows required by your program, generally the only other thing that the **main()** function does is receive and dispatch messages. To accomplish this it uses a loop that looks something like

```
while(program is still running) {
        get a message;
        send the message to the proper window;
}
```

Essentially, the Presentation Manager communicates with your program by putting messages into its message queue. Your program then extracts a message from the queue and dispatches it to the proper window by calling another Presentation Manager service. This process continues until the program is terminated. For the most part messages are the only way in which your program receives input. (Remember that a Presentation Manager program cannot, for example, call **scanf()** to read input from the keyboard.) Although the form of a message varies somewhat depending on what type of message it is, all messages are integers.

A CLOSER LOOK AT A WINDOW

All Presentation Manager windows begin with a *frame*, which is essentially a box. A number of optional but desirable additions are made to this frame. In OS/2 these additional features are actually windows in their own right. However, it is easier to think of them as options to the frame. Let's look at these options now.

Two "options" are vitally necessary for all windows. The first is the *border*. The border is important because it allows the user to move or resize the window using the mouse. The second is the *system menu*. The system menu is a standard menu that, at a minimum, allows the user to perform the following operations: restore the window to its original size, move the window, resize the window, minimize or maximize the window, and close the window. Although the border allows a more convenient method of moving or resizing the window, these operations can also be activated from the system menu. When a window is minimized, it is shown in its iconic form and is moved to the icon region of the screen. Your program can specify what the iconic form of a window will look like or simply let the system decide. When a window is maximized, it takes over the entire screen. Closing a window removes it from the screen and, if this is the program's top-level window, terminates the program.

Most of the time you will also want to add three other features to your windows: the ability to maximize and minimize icons and a title that identifies the window. Although it is possible to maximize and minimize the window by using the system menu, it is quicker to use maximize and minimize icons because the user can activate them with the mouse. When the screen holds several windows, titles remind the user which window is which.

Finally, you will add vertical and horizontal scroll bars to the window if your program needs them. By clicking on appropriate points in these scroll bars the user scrolls the contents of the window up, down, left, or right.

The region enclosed by the frame and used by your application program is called the *client area*.

The organization of a standard Presentation Manager window appears in Figure 11-1. (Remember that not all options are necessarily used for all windows.)

Each Presentation Manager-compatible program creates one or more main windows. A *main window* is at the topmost level and is the

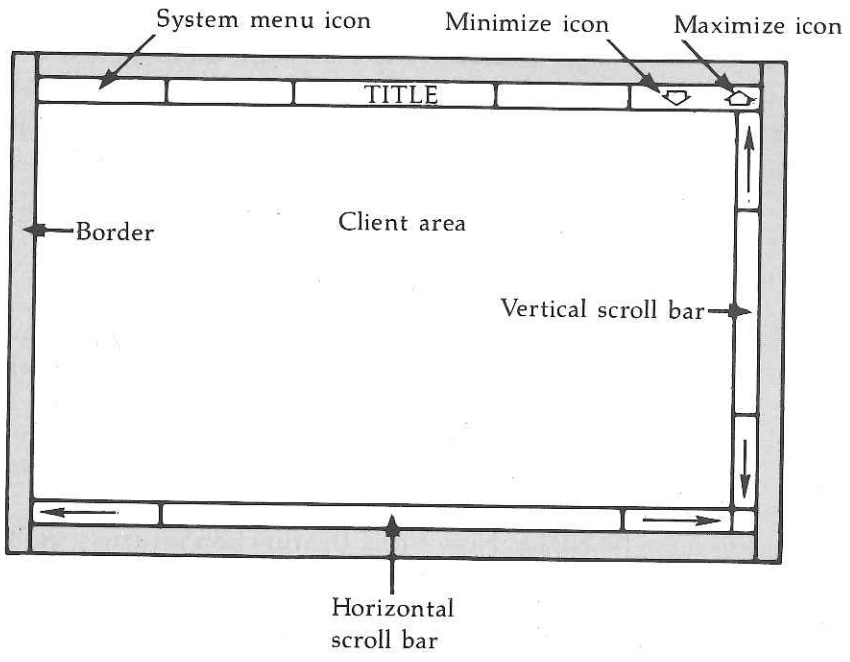


Figure 11-11. The layout of a standard window

window that the user associates with the program. Closing the main window terminates the program.

There are two general categories of windows: parents and children. When an application begins, it creates one or more main windows. If it creates more than one main window, they overlap each other. However, it is possible to create a window inside another window. In this case the newly created window is a child of the main window and is enclosed by the parent. A child window can, in turn, create a child of its own, and so on up to the limits imposed by the size of the screen.

Each window is associated with a class. There are several built-in classes, such as menus, frames, scroll bars, and the like. However, windows that you create need to be given class names, and these classes must be registered with the Presentation Manager.

All windows define the lower left corner as location 0,0. The maximum *x* and *y* dimensions are dynamically defined as the window changes size and shape. However, the maximum dimension is determined by the resolution of the screen.

Now that you know some of the theory behind the Presentation Manager and its windows, let's look at some specifics.

OBTAINING AN ANCHOR BLOCK USING WinInitialize

One of the first things that you will want your Presentation Manager application to do is obtain an *anchor block handle* by calling **WinInitialize**, whose prototype is

```
void far *WinInitialize(unsigned short handle);
```

Here, *handle* must be **NULL**. Notice that the function returns a **void far** pointer, which points to the region of memory used by the Presentation Manager to hold various bits of information about the window environment relative to the application program. This region of memory is called the anchor block and the pointer to it is called the anchor block handle. If the system cannot be initialized, a **NULL** is returned. The anchor block handle is required as a parameter by many Presentation Manager services.

Unlike the core API services, which return 0 for success, many of the Presentation Manager services return 0 (**NULL**) on failure.

CREATING A MESSAGE QUEUE

After initializing the window system all Presentation Manager applications must create a message queue by using **WinCreateMsgQueue**, which has the prototype

```
void far *WinCreateMsgQueue(void far *anchor_block, int size);
```

where *anchor_block* is the handle obtained by using **WinInitialize**. The size of the queue is determined by the value of *size* or, if *size* is **NULL**, the system default is used. Generally the system default queue size is acceptable.

Each element in the message queue is contained in a structure (called **QMSG** by Microsoft) defined as

```
struct {
    void far *hwnd;          /* handle of the recipient window */
    unsigned short msg;      /* the message */
    void far *mp1;           /* additional message info */
    void far *mp2;           /* additional message info */
    unsigned long time;      /* time message was generated */
    POINTL ptl;              /* position of mouse pointer */
} QMSG;
```

The **POINTL** structure is defined as

```
struct {
  long x;
  long y;
} POINTL;
```

WinCreateMsgQueue returns a handle to the message queue or **NULL** if the request fails.

REGISTERING A WINDOW CLASS

Before you can actually create a window, you must register its class using **WinRegisterClass**, which has the prototype

```
unsigned short WinRegisterClass(void far *anchor_block,
                                char far *classname,
                                (pascal far * window_func) (),
                                unsigned long style,
                                int storage_bytes);
```

where *anchor_block* is a pointer to the anchor block. The string pointed to by *classname* is the name of the window class being registered. The address of the window function must be passed as the third parameter. The style of the window is specified by *style*. The number of bytes of additional storage beyond that needed by the window is specified by

storage_bytes. Your program may use this extra storage for its own purposes. In the examples in this book, this field will be 0.

The value of *style* describes the sort of window being registered. The only style used in this book has the value 4L and is defined as **CS_SIZEREDRAW** in the PMWIN.H header file provided by Microsoft. Using this style causes the Presentation Manager to inform your program each time the window is resized.

The **WinRegisterClass** service returns nonzero if successful and **NULL** if unsuccessful.

CREATING A STANDARD WINDOW

Once you have initialized the window system relative to your application, created a message queue, and registered the class, it is time to create a window. The easiest way to create a standard window is to use **WinCreateStdWindow**, which has the prototype

```
void far *WinCreateStdWindow(void far *anchor_block,
                             void far *parent_handle,
                             unsigned long style;
                             char far *classname,
                             char far *title,
                             unsigned module,
                             unsigned long client_style,
                             int resource,
                             void far **client_handle);
```

The *parent_handle* must be the handle of the parent window. When a program begins execution, its parent is the screen, which has the handle 1. Microsoft defines this value by the macro **HWND_DESKTOP**. This value will be used for the examples in this chapter.

The value of *style* determines several features of the window. It can be a combination of several values. The most common, along with the macro names given to them by Microsoft, appear in Table 11-1.

The *classname* parameter points to the string that identifies the class. This should be the same string that was used in the call to **WinRegisterClass**.

The string pointed to by *title* is used as the title of the window for

Table 11-1. The Most Common Values for the **WinCreateStdWindow** Style Parameter

Macro Name	Value	Meaning
WS_VISIBLE	0x80000000L	Make window visible
WS_MINIMIZED	0x01000000L	Minimize window
WS_MAXIMIZED	0x00800000L	Maximize window
FS_TITLEBAR	0x00000001L	Include title bar
FS_SYSMENU	0x00000002L	Include system menu
FS_VERTSCROLL	0x00000010L	Include vertical scroll bar
FS_HORZSCROLL	0x00000020L	Include horizontal scroll bar
FS_SIZEBORDER	0x00000040L	Include sizing border
FS_BORDER	0x00000200L	Use thin border
FS_MINBUTTON	0x00001000L	Include minimize icon
FS_MAXBUTTON	0x00002000L	Include maximize icon
FS_MINMAX	0x00003000L	Include both minimize and maximize icons

identification purposes.

For most purposes the *client_style* parameter should be 0L, indicating that the client window should be the same style as the window class.

The *resource* and *module* parameters identify a resource module. The examples in this chapter need no resource modules, so these parameters should be **NULL** and 0 respectively.

The **WinCreateStdWindow** service returns a handle in *client_handle* to the frame if successful and **NULL** if not.

THE MESSAGE LOOP

To receive messages your program will need to use **WinGetMsg**, which has the prototype

```
unsigned short WinGetMsg(void far *anchor_block,
                         QMSG far *message,
                         void far *window,
                         unsigned short first,
                         unsigned short last);
```

The message retrieved from the queue is put in the queue structure pointed to by *message*. If *window* is not null, it causes **WinGetMsg** to retrieve only the messages directed to the specified window. Most of the time your application will want to receive all messages. In this case *window* should be **NULL**. All messages are integers. The *first* and *last* parameters determine the range of messages that will be accepted by defining the end points of that range. If you wish to receive all messages, *first* and *last* should both be 0. The **WinGetMsg** service returns true unless a termination message is received, in which case it returns false.

In many situations, once a message has been received it is simply dispatched to the correct window without further processing by your program within the message loop. The service that sends messages along their way is **WinDispatchMsg**, which has the prototype

```
void far *WinDispatchMsg(void far *anchor_block,
                         QMSG far *message);
```

When you call this function the message is automatically routed to the proper window function. **WinDispatchMsg** returns the value returned by the window function.

PROGRAM TERMINATION

Before your program terminates, it must do three things: close any active windows, close the message queue, and deactivate the window system interface created by the **WinInitialize** service. To accomplish these things the Presentation Manager provides the services **WinDestroyWindow**, **WinDestroyMsgQueue**, and **WinTerminate**, which have the prototypes

```
unsigned long WinDestroyWindow(void far *handle_window);
unsigned long WinDestroyMsgQueue(void far *handle_msgQ);
unsigned long WinTerminate(void far *anchor_block);
```

Here *handle_window* is the handle of the window to be closed. The *handle_msgQ* is the handle to the message queue to be destroyed. Finally the window system is disconnected by calling **WinTerminate** with the anchor block handle.

THE WINDOW FUNCTION

As mentioned earlier, all programs that are compatible with the Presentation Manager must pass to the Presentation Manager the address of the window function that will receive messages. This function must be declared as shown here:

```
void far * pascal far window_func(void far *handle,
                                  unsigned short message,
                                  void far *param1,
                                  void far *param2);
```

The window function receives the Presentation Manager messages in its parameters. In essence the Presentation Manager sends your program a message by calling the window function. The value of *handle* is the handle of the window receiving the message. The integer *message* contains the message itself. Some messages require further information, which is put into the *param1* and *param2* parameters.

The Presentation Manager can generate several different types of messages. Some of the more common ones appear in Table 11-2 along with the macro names assigned to them by Microsoft. Some of these messages will be used in the sample programs developed in this chapter and the next.

The window function does not need explicitly to process all the messages that it receives. In fact an application commonly processes only a few types of messages. What happens, then, to the rest of the messages received by the window function? They are passed back to the Presentation Manager for default processing using the **WinDefWindowProc** service which has the prototype

```
void far *WinDefWindowProc(void far *handle,
                           unsigned short message,
                           void far *param1,
                           void far *param2);
```

Table 11-2. Some Common Messages

Macro Name	Value	Meaning
WM_BUTTON1DOWN	0x0071	Button 1 down
WM_BUTTON1UP	0x0072	Botton 1 up
WM_BUTTON1DBLCLK	0x0073	Double click on button 1
WM_BUTTON2DOWN	0x0074	Button 2 down
WM_BUTTON2UP	0x0075	Button 2 up
WM_BUTTON2DBLCLK	0x0076	Double click on button 2
WM_BUTTON3DOWN	0x0077	Button 3 down
WM_BUTTON3UP	0x0078	Button 3 up
WM_BUTTON3DBLCLK	0x0079	Double click on button 3
WM_CHAR	0x007A	Keystroke occurred
WM_CREATE	0x0001	Window has been created
WM_DESTROY	0x0002	Window is being destroyed
WM_ERASEBACK-GROUND	0x004F	OK to erase background request
WM_HSCROLL	0x0032	Horizontal scroll
WM_MOVE	0x0006	Window is being moved
WM_MOUSEMOVE	0x0070	Mouse has moved
WM_PAINT	0x0023	Window display needs to be refreshed
WM_SHOW	0x0005	Window is shown or removed from the screen
WM_SIZE	0x0007	Window is being resized
WM_VSCROLL	0x0031	Vertical scroll
WM_QUIT	0x002A	Window is being terminated

As you can probably guess, **WinDefWindowProc** simply passes back to the Presentation Manager the parameters with which it was called.

PUTTING TOGETHER THE PIECES: A PRESENTATION MANAGER SKELETON PROGRAM

Now that you have seen the services needed to initialize and run a simple windowed application, it is time to see some real code! The fol-

lowing skeleton program creates a window that includes a system menu, a title, a sizing border, and scroll bars. You can move the window about the screen, minimize or maximize it, change its shape, and terminate it—nothing else. For the moment don't worry too much about the window function **window_func()**; it will be explained shortly.

```
/* A Presentation Manager Application skeleton. */

#define INCL_WIN

#include <os2.h>
#include <stddef.h>  /* get definition of NULL */

void far * pascal far window_func(void far *, unsigned short,
                                  void far *, void far *);

char class[] = "MyClass";

main()
{
  void far *hand_ab;
  void far *hand_mq;
  void far *hand_frame, far *hand_client;
  QMSG q_mess;

  hand_ab = WinInitialize(NULL);

  hand_mq = WinCreateMsgQueue(hand_ab, 0);

  if(!WinRegisterClass(hand_ab,    /* anchor block */
                       class,      /* class name */
                       window_func,/* address of window function */
                       CS_SIZEREDRAW, /* window style */
                       0))         /* no storage reserved */
    exit(1);

  hand_frame = WinCreateStdWindow(HWND_DESKTOP,
                  WS_VISIBLE | FS_SYSMENU |
                  FS_SIZEBORDER | FS_TITLEBAR |
                  FS_VERTSCROLL| FS_HORZSCROLL |
                  FS_MINMAX,
                  (char far *) class,
                  (char far *) "My Window",
                  0L,   /* resource modules */
                  NULL,
                  0,
                  &hand_client); /* client handle */

  /* message loop */
  while(WinGetMsg(hand_ab, &q_mess, NULL, 0, 0))
    WinDispatchMsg(hand_ab, &q_mess);

  WinDestroyWindow(hand_frame);

  WinDestroyMsgQueue(hand_mq);
  WinTerminate(hand_ab);
}

/* This is the window function. */
void far * pascal far window_func(void far *handle,
                                  unsigned short mess,
                                  void far *parm1,
                                  void far *parm2)
```

```
{
  switch(mess) {
      case WM_CREATE:
        /* Perform any necessary initializations here. */
      break;

    case WM_PAINT:
        /* Refresh the window each time the WM_PAINT message
           is received.
        */
      break;

    case WM_ERASEBACKGROUND:
        /* By returning TRUE, the PM automatically erases
           the old window each time the window is resized
           or moved.   Without this, your program must
           manually handle erasing the window when it changes
           size or location.
        */
      return(TRUE);

    case WM_CHAR:
        /* Process keystrokes here. */
      break;

    case WM_HSCROLL:
        /* Process horizontal scroll request.  */
      break;

    case WM_VSCROLL:
        /* Process vertical scroll request. */
      break;

    case WM_MOUSEMOVE:
        /*  Process a mouse motion message. */
      break;

    case WM_BUTTON1DOWN:
        /* 1st mouse button is pressed. */
      break;

    case WM_BUTTON2DOWN:
        /* 2nd mouse button is pressed. */
      break;

    case WM_BUTTON3DOWN:
        /* 3rd mouse button is pressed. */
      break;
    /* If required by your application, you may also need to
       process these mouse messages:

            WM_BUTTON1UP
            WM_BUTTON1DBLCLK
            WM_BUTTON2UP
            WM_BUTTON2DBLCLK
            WM_BUTTON3UP
            WM_BUTTON3DBLCLK
    */

    default:
      /* All messages not handled by the window_func,
         must be passed along to the PM for default
         processing.
```

```
        */
        return WinDefWindowProc(handle, mess, parm1, parm2);
    }
    return 0L;
}
```

Note that the program defines **INCL_WIN**. This is necessary to include the prototypes and definitions for the window system on OS2.H.

Read the next two sections before you try to compile this program.

The Definition File

Unlike non-Presentation Manager programs, any Presentation Manager-compatible program you write needs to include a definition file in the link line. Other reasons aside, you will need to specify more stack space for the Presentation Manager application than it will receive by default. The Presentation Manager examples in this book allocate 4096 bytes, but real-world applications may need more space. It is also a good idea to specify a heap size. The programs in this book allocate 4096 bytes for this purpose, but your programs may need more or less. You must also include an EXPORTS statement in the file that specifies the name of the window function. The definition file for the skeleton just shown looks like this:

```
NAME skeleton
HEAPSIZE 4096
STACKSIZE 4096
EXPORTS window_func
```

Compiling Presentation Manager Programs

You will need to specify some different compiler options for a Presentation Manager program than for a standard program. You can use this batch file if you are using the Microsoft C compiler:

```
CL -c -G2sw %1.c
LINK %1,,, os2, %1.def;
RC %1
```

The -G2sw option tells the compiler to use 32-bit addresses for all code and data references, turn off stack checking, assume that the value of

the **DS** register is different from the value of the **SS** register, and generate 80286 instructions. Since the Presentation Manager requires at least an 80286 processor, there is no harm in generating 80286 instructions.

Note that the link line specifies the library called OS2.LIB instead of DOSCALLS.LIB. This name was introduced with version 1.1 of OS/2. However, different versions of OS/2 may call this something else.

Note: You may have to use a different set of options even if you are using Microsoft C because of future changes to the compiler. Be sure to read your instruction manual carefully on this point.

Understanding How the Skeleton Works

The operation of the **main()** function is straightforward. It initializes the link between the Presentation Manager and the program, registers a new window class, creates a window, and executes its message loop. As messages are received, they are dispatched to the **window_func()** by calling **WinDispatchMsg**. The message loop terminates when the **WM_QUIT** message is received. This message is generated by choosing the **close** option in the window's system menu.

The most important single function in a Presentation Manager application is the window function. It receives the messages sent by the Presentation Manager and takes appropriate action. The skeleton shows entries in the **switch** statement for only the most common of the several messages that can be generated by the Presentation Manager. (Remember that any message your program does not wish to process must be passed back to the Presentation Manager via the **WinDefWindowProc** service.) Let's look at the meaning of some of these.

When a window is created, the **WM_CREATE** message is sent to the window function. This allows your program to initialize values or perform other startup operations.

The Presentation Manager allows the user to move and resize windows and also to cover part of a window with another. These operations imply that all or part of the window must be redrawn at some time. The Presentation Manager generates the **WM_PAINT** message whenever the contents of the window must be refreshed.

The **WM_ERASEBACKGROUND** message tells your program that the window needs to be erased, perhaps because the window is being

moved. By returning TRUE, you allow the Presentation Manager to do this for you. Otherwise, your program must do it.

Each time the user presses a key, the **WM_CHAR** message is generated. This message will be discussed further in the next chapter.

Each time the user requests a vertical scroll the **WM_VSCROLL** message is generated. Each time the user requests a horizontal scroll, the **WM_HSCROLL** message is generated.

The mouse messages are self-explanatory.

Because this program is a skeleton for future applications, it does not do anything with the messages. However, you will soon see examples that do. Keep in mind that when your program does not actually need to worry about a message—if the program does not have scroll bars, for example—that message can be removed from the **switch** statement and the default processing will handle it.

PRESENTATION MANAGER VERSUS CORE SERVICES

At this point you might be wondering how the Presentation Manager services relate to the core OS/2 services. There certainly appears to be significant overlap in many areas. In general, if you wish to write Presentation Manager-compatible programs that follow the standard Presentation Manager style, you must not use any of the **Vio**, **Kbd**, or **Mou** services in your program. Instead you must use the comparable Presentation Manager services. However, feel free to use the **Dos** core services, especially those that support interprocess communication and device monitors.

If a program uses a **Vio**, **Kbd**, or **Mou** service, it will be run in its own screen group, not in a Presentation Manager window.

The main use for the **Vio**, **Kbd**, or **Mou** services is for utility programs, especially programmer utilities, that do not need the support of a windowed environment.

12

SOME PRESENTATION MANAGER EXAMPLES

This chapter introduces some of the commoner programming tasks, such as input, output, and the use of graphics and menus, as they are formulated in a Presentation Manager environment. While it is beyond the scope of this book to go into significant detail about the Presentation Manager, the material presented in this chapter will help you grasp some of the more important Presentation Manager programming concepts and will provide a base for further study.

OUTPUTTING TEXT

Outputting text to a client window is not as easy as you might expect for two reasons:

1. You can't use any of the C run-time functions such as **printf()**.
2. You can't use any of the VIO API services either!

The reason for these restrictions is that neither the C standard output functions nor the VIO functions have any knowledge of a windowed environment.

Beyond the fact that your programs must use special Presentation Manager output functions to display text in a window, it is still not a trivial task to output text because the Presentation Manager maintains a level of abstraction between your program and the output device.

Before developing any examples, you need to learn a few new terms and concepts.

Presentation Space and Device Context

When your program outputs something to the "screen," it is actually outputting information to a *presentation space* (PS), which you can think of as being a data structure that contains several pieces of information about the size and form of the "screen." The reason that the word *screen* has been placed in quotes in the foregoing sentences is that a presentation space is not necessarily linked to the screen; it could be linked with the printer, for example. The device that the presentation space is actually linked to is called the *device context* (DC). For the rest of this discussion, the device context is assumed to be the screen.

There are three types of presentation spaces: the normal-PS, the micro-PS, and the cached micro-PS. The examples in this chapter use only the cached micro-PS, but it is important that you understand the general concept behind all three.

The normal-PS is the most flexible of the three presentation spaces. Your program uses it when it writes to devices other than the screen or when a screen display is in existence a long time without a refresh. A micro-PS is similar to a normal-PS except that it requires less memory and has fewer capabilities. The cached micro-PS is the simplest presentation space to use and requires the least memory. However, the cached micro-PS operates only with the screen, so it cannot be used to send output to any other device.

Processing the WM_PAINT Message

As you probably recall from the previous chapter, each time a window is moved, resized, or uncovered, the **WM_PAINT** message is sent to the window function. Each time this message is received, your program must completely redisplay any output that was in the window. The process is often called *refreshing* the window. Although it is possible to output to the window during the processing of other messages, the most common time for this to occur is when handling the **WM_PAINT** message. For this reason the discussion of text output begins with how it relates to the processing of the **WM_PAINT** message.

Before you can output anything to the screen, you need to obtain a presentation space handle. There are several ways to do this, but when processing the **WM_PAINT** message the easiest way is to use the **WinBeginPaint** service to return a micro-PS handle. The prototype for **WinBeginPaint** is

```
void far *WinBeginPaint(void far *handle,
                        void far *p_space,
                        RECTL far *region);
```

where *handle* is the handle of the window that will be drawn to, and *p_space* is the handle of the presentation space. If this value is **NULL**, a micro-PS is automatically allocated and its handle returned by the service. The structure pointed to by *region* contains the coordinates of the region that needs to be updated. This parameter may be **NULL** in cases where it is simply easier for the program to update the entire window rather than a portion.

WinBeginPaint has a second important function: It informs the Presentation Manager that a window refresh is beginning. For this reason it is a good idea to call **WinBeginPaint** immediately after receiving a **WM_PAINT** message.

The simplest way to write a line of text to a window is to use the **GpiCharStringAt** service, whose prototype is

```
long GpiCharStringAt(void far *p_space,
                     POINTL far *loc,
                     long size,
                     char far *string);
```

where *p_space* is the presentation space handle. The structure pointed to by *loc* contains the coordinates of the location at which the string will be written. The *size* parameter holds the size, and *string* points to the actual string.

The return value of **GpiCharStringAt** is somewhat complex and is not required by the examples in this chapter.

The **POINTL** structure is defined like this:

```
struct POINTL {
  long x;
  long y;
} ;
```

It is critical to keep in mind that the *x,y* locations in the **POINTL** structure are specified in pels, not in characters.

Although in its default mode no cursor is seen in a window, each window does keep track of the position of an invisible "cursor." The position of this invisible cursor is called the *current position*. Many of the output services, including **GpiCharStringAt**, affect the location of the current position. After the string has been displayed by using **GpiCharStringAt**, the current position is advanced to the pel immediately following the last character in the string.

The **GpiCharStringAt** service does not process carriage returns or linefeeds, so your program must manually advance to new lines when needed.

Before the code that processes the **WM_PAINT** message finishes, it must issue a call to **WinEndPaint**, which has the prototype

```
unsigned short WinEndPaint(void far *p_space);
```

where *p_space* is the handle of the presentation space updated by the program. If **WinEndPaint** is successful, it returns true; otherwise it returns false.

Assuming the necessary variable declarations, the following fragment outputs "This is a test" on the screen starting at the lower left corner:

```
case WM_PAINT:
  /* get a handle to the presentation space */
  p_space = WinBeginPaint(handle, NULL, NULL);

  /* output a message that starts at the lower
     left corner
  */
  coords.x = 0L;
  coords.y = 0L;
  GpiCharStringAt(p_space, (POINTL far *) &coords,
                  14L,
                  (char far *) "This is a test");
  /* close the presentation space */
  WinEndPaint(handle);
  break;
```

Each time the window associated with this code fragment is moved, resized, or uncovered, the **WM_PAINT** message is received and the line of text is redisplayed. An entire program that uses the code fragment is shown here:

```
/* Output a string. */

#define INCL_WIN
#define INCL_GPI

#include <os2.h>
#include <stddef.h>  /* get definition of NULL */

void far * pascal far window_func(void far *, unsigned short,
                          void far *, void far *);

char class[] = "MyClass";

main()
{
  void far *hand_ab;
  void far *hand_mq;
  void far *hand_frame, far *hand_client;
  QMSG q_mess;

  hand_ab = WinInitialize(NULL);

  hand_mq = WinCreateMsgQueue(hand_ab, 0);

  if(!WinRegisterClass(hand_ab,      /* anchor block */
                  class,        /* class name */
                  window_func, /* address of window function */
                  CS_SIZEREDRAW, /* window style */
                  0))           /* no storage reserved */
     exit(1);

  hand_frame = WinCreateStdWindow(HWND_DESKTOP,
                  WS_VISIBLE | FS_SYSMENU |
                  FS_SIZEBORDER | FS_TITLEBAR |
                  FS_MINMAX,
                  (char far *) class,
                  (char far *) "My Window",
                  0L,
                  NULL,
                  0,
                  &hand_client);


  while(WinGetMsg(hand_ab, &q_mess, NULL, 0, 0))
    WinDispatchMsg(hand_ab, &q_mess);

  WinDestroyWindow(hand_frame);

  WinDestroyMsgQueue(hand_mq);
  WinTerminate(hand_ab);
}

/* window function */
void far * pascal far window_func(void far *handle,
                                  unsigned short mess,
                                  void far *parm1,
                                  void far *parm2)
{
  void far *p_space;
  POINTL coords;

  switch(mess) {
```

```
    case WM_PAINT:
      /* get a handle to the presentation space */
      p_space = WinBeginPaint(handle, NULL, NULL);

      /* output a message that starts at the lower
         left corner
      */
      coords.x = 0L;
      coords.y = 0L;
      GpiCharStringAt(p_space, (POINTL far *) &coords,
                      14L,
                      (char far *) "This is a test");
      /* close the presentation space */
      WinEndPaint(handle);
      break;
  case WM_ERASEBACKGROUND:
      return(TRUE);

    default:
      return WinDefWindowProc(handle, mess, parm1, parm2);
  }
  return 0L;
}
```

To compile this program be certain to use the method discussed in the previous chapter and include a definition file similar to the following (in fact, be sure to include a similar definition file with all the sample programs in this chapter):

```
NAME prog_name
HEAPSIZE 4096
STACKSIZE 4096
EXPORTS window_func
```

Displaying Text in Color

You can change both foreground and background colors by using **GpiSetColor** and **GpiSetBackColor**, respectively. Their prototypes are

```
unsigned short GpiSetColor(void far *p_space,
                           long color);
unsigned short GpiSetBackColor(void far *p_space,
                               long color);
```

Here *p_space* is the handle to the presentation space and *color* is the desired color, which can be one of these values (shown along with the macro names given them by Microsoft):

Macro Name	Value
CLR_DEFAULT	−3L
CLR_WHITE	−2L
CLR_BLACK	−1L
CLR_BACKGROUND	0L
CLR_BLUE	1L
CLR_RED	2L
CLR_PINK	3L
CLR_GREEN	4L
CLR_CYAN	5L
CLR_YELLOW	6L
CLR_NEUTRAL	7L
CLR_DARKGRAY	8L
CLR_DARKBLUE	9L
CLR_DARKRED	10L
CLR_DARKPINK	11L
CLR_DARKGREEN	12L
CLR_DARKCYAN	13L
CLR_BROWN	14L
CLR_LIGHTGRAY	15L

Keep in mind that once you set a foreground or background color, or both, they remain in effect until reset.

In the Presentation Manager's default mode of operation, once the foreground color is set, all subsequent screen output operations take place in that color. This is not the case for the background color, however, because by default the new background color is not "mixed" into the background color of the window. In order to mix the color in, you must call the **GpiSetBackMix** service, whose prototype is

```
unsigned short GpiSetBackMix(void far *p_space,
                             long mix);
```

where *p_space* is the presentation space of the window and *mix* is the value that determines how the background color is mixed with the current screen color. The most common values are shown here along with the macro names defined by Microsoft.

Macro Name	Value	Meaning
BM_DEFAULT	0L	Use system default.
BM_OVERPAINT	2L	Overwrite current color.
BM_LEAVEALONE	5L	Leave current background color unchanged.

You use **BM_OVERPAINT** to have the background color replace the current screen color.

Although it is not used by the sample programs in this chapter, you can set the mix of the foreground color by using **GpiSetMix**, whose prototype is

```
unsigned short GpiSetMix(void far *p_space,
                         long mix);
```

Here *mix* specifies how the foreground color will be displayed. The most common values are

Name	Value	Meaning
FM_DEFAULT	0L	Use default.
FM_OR	1L	OR text onto screen.
FM_OVERPAINT	2L	Overwrite current screen color.
FM_LEAVEALONE	5L	Leave color attributes unchanged.
FM_XOR	4L	XOR text onto screen.
FM_AND	6L	AND text onto screen.

You may want to experiment with this service on your own.

The following program uses **GpiSetBackColor**, **GpiSetColor**, and **GpiSetBackMix** to display a string using blue foreground and red background.

```
/* Output blue text on red background.
*/
#define INCL_WIN
#define INCL_GPI

#include <os2.h>
#include <stddef.h>  /* get definition of NULL */

void far * pascal far window_func(void far *, unsigned short,
                          void far *, void far *);
```

```
char class[] = "MyClass";

main()
{
  void far *hand_ab;
  void far *hand_mq;
  void far *hand_frame, far *hand_client;
  QMSG q_mess;

  hand_ab = WinInitialize(NULL);

  hand_mq = WinCreateMsgQueue(hand_ab, 0);

  if(!WinRegisterClass(hand_ab,    /* anchor block */
                (char far *) class, /* class name */
                window_func,    /* address of window function */
                CS_SIZEREDRAW, /* window style */
                0))             /* no storage reserved */
     exit(1);

  hand_frame = WinCreateStdWindow(HWND_DESKTOP,
                WS_VISIBLE | FS_SYSMENU |
                FS_SIZEBORDER | FS_TITLEBAR |
                FS_VERTSCROLL| FS_HORZSCROLL |
                FS_MINMAX,
                (char far *) class,
                (char far *) "My Window",
                0L,
                NULL,
                0,
                &hand_client); /* client handle */


  while(WinGetMsg(hand_ab, &q_mess, NULL, 0, 0))
    WinDispatchMsg(hand_ab, &q_mess);

  WinDestroyWindow(hand_frame);

  WinDestroyMsgQueue(hand_mq);
  WinTerminate(hand_ab);
}

void far * pascal far window_func(void far *handle,
                                  unsigned short mess,
                                  void far *parm1,
                                  void far *parm2)
{
  void far *p_space;
  POINTL coords;

  switch(mess) {
    case WM_PAINT:
      /* get a presentation space handle */
      p_space = WinBeginPaint(handle, NULL, NULL);

      /* use red background */
      GpiSetBackColor(p_space, CLR_RED);
      /* set mix to overwrite */
      GpiSetBackMix(p_space, BM_OVERPAINT);
      /* set foreground to blue */
      GpiSetColor(p_space, CLR_BLUE);
```

```
        coords.x = 0L;
        coords.y = 0L;
        GpiCharStringAt(p_space, (POINTL far *) &coords,
                        14L,
                        (char far *) "This is a test");
        /* close the presentation space */
        WinEndPaint(handle);

        break;

    case WM_ERASEBACKGROUND:
        /* By returning TRUE, the PM automatically erases
           the old window each time the window is resized
           or moved.   Without this, your program must
           manually handle erasing the window when it changes
           size or location.
        */
        return(TRUE);

    default:
        /* All messages not handled by the window_func,
           must be passed along to the PM for default
           processing.
        */
        return WinDefWindowProc(handle, mess, parm1, parm2);
  }
  return 0L;
}
```

WinGetPS and WinReleasePS

You can obtain a handle to a cached micro-PS without using **WinBeginPaint** by using **WinGetPS**, whose prototytpe is

```
void far *WinGetPS(void far *win_handle);
```

Here, *win_handle* is the handle of the window to which you will be outputting. The handle to the presentation space is returned by the service.

Since you can call **WinBeginPaint** only when the **WM_PAINT** message is received, the **WinGetPS** service is useful when you want to output information during the processing of another message. (An example of this appears in the next section.)

When your routine has finished outputting, it must call **WinReleasePS**, which has the prototype

```
unsigned short WinReleasePS(void far *p_space);
```

where *p_space* is the presentation space handle obtained by a call to **WinGetPS**.

READING KEYSTROKES

As mentioned in passing in the previous chapter, your Presentation Manager programs cannot read keyboard input in the traditional fashion. For example, your programs cannot call such standard library functions as **gets()** or **scanf()**. Instead, each time a key is pressed a **WM_CHAR** message is sent to the active window.

The keystroke information is encoded in the two message parameters as follows. The first 16 bits of the first parameter contain several flags that tell you what type of key was pressed. The flags are encoded, as shown here (along with their macro names defined by Microsoft):

Macro Name	Value	Meaning When Set
KC_CHAR	1	Is character
KC_VIRTUALKEY	2	Is special key
KC_SCANCODE	4	Is scan code
KC_SHIFT	8	Is SHIFT key
KC_CTRL	16	Is CONTROL key
KC_ALT	32	Is ALT key
KC_KEYUP	64	Key is being released
KC_PREVDOWN	128	Key was down
KC_LONEKEY	256	Is single key
KC_DEADKEY	512	Is unused key
KC_COMPOSITE	1024	Is key combination
KC_INVALIDCOMP	2048	Is invalid combination
KC_TOGGLE	4096	Is toggle key

The next 8 bits of the first parameter give a repetition count. This indicates how many times the key has been autorepeated. Generally you will not need to worry about this.

The high-order 8 bits of the first parameter hold the key's scan code. As you probably remember from Chapter 4, when you press a key, OS/2 generates a scan code, which, in the case of normal keys, is associated with a character code. Certain keys, however, such as the arrow keys, do not have character codes, which means that the scan code is used to identify them. (Refer to Chapter 4 for more information on scan and character codes.)

The second parameter associated with the **WM_CHAR** message contains two items. The lower 16 bits contain the character code, assuming that a regular key has been pressed. That is, if the **KC_CHAR** flag is set in the first parameter, a valid character code is found

in the lower 16 bits of the second parameter. However, if you press a special key, the **KC_CHAR** flag is not set and the character code of the second parameter is 0. For U.S.-style keyboards only the first 8 bits are of interest, but for foreign systems the full 16 bits may be needed.

The high-order 16 bits of the second parameter hold the virtual key code for the key that was pressed. All keystrokes are assigned a virtual code. For normal keys, however, this code is 0. The virtual key codes, along with their corresponding macro names (defined by Microsoft) are shown in Table 12-1. As you can see, some virtual key codes cannot be generated by the keyboard, but are generated by the Presentation Manager itself.

Table 12-1. The Virtual Key Codes

Macro Name	Value	Key
VK_CANCEL	04	CANCEL
VK_BACK	05	BACKSPACE
VK_TAB	06	TAB
VK_CLEAR	07	
VK_RETURN	08	ENTER
VK_SHIFT	09	SHIFT
VK_CONTROL	10	CONTROL
VK_ALT	11	ALT
VK_ALTGRAF	12	
VK_PAUSE	13	PAUSE
VK_CAPITAL	14	CAPS LOCK
VK_ESCAPE	15	ESCAPE
VK_SPACE	16	SPACE
VK_PGUP	17	PAGE UP
VK_PGDN	18	PAGE DOWN
VK_END	19	END
VK_HOME	20	HOME
VK_LEFT	21	LEFT ARROW
VK_UP	22	UP ARROW
VK_RIGHT	23	RIGHT ARROW
VK_DOWN	24	DOWN ARROW
VK_SELECT	25	
VK_PRINT	26	
VK_EXECUTE	27	
VK_INSERT	28	INS

Table 12-1. The Virtual Key Codes *(continued)*

Macro Name	Value	Key
VK_DELETE	29	DEL
VK_SCRLLOCK	30	SCROLL LOCK
VK_NUMLOCK	31	NUM LOCK
VK_NUMPAD0	32	Number pad 0
VK_NUMPAD1	33	Number pad 1
VK_NUMPAD2	34	Number pad 2
VK_NUMPAD3	35	Number pad 3
VK_NUMPAD4	36	Number pad 4
VK_NUMPAD5	37	Number pad 5
VK_NUMPAD6	38	Number pad 6
VK_NUMPAD7	39	Number pad 7
VK_NUMPAD8	40	Number pad 8
VK_NUMPAD9	41	Number pad 9
VK_ADD	42	Number pad +
VK_SUBTRACT	43	Number pad −
VK_MULTIPLY	44	Number pad *
VK_DIVIDE	45	Number pad /
VK_DECIMAL	46	Number pad .
VK_ENTER	47	Number pad enter
VK_F1	48	F1
VK_F2	49	F2
VK_F3	50	F3
VK_F4	51	F4
VK_F5	52	F5
VK_F6	53	F6
VK_F7	54	F7
VK_F8	55	F8
VK_F9	56	F9
VK_F10	57	F10
VK_F11	58	F11
VK_F12	59	F12
VK_F13	60	F13
VK_F14	61	F14
VK_F15	62	F15
VK_F16	63	F16
VK_HELP	64	
VK_SYSREQ	65	SysRq
VK_MENU	11	Same as VK_ALT
VK_INS	28	Same as VK_INSERT
VK_DEL	29	Same as VK_DELETE

As you saw in Chapter 4, each time you press a key, OS/2 generates a *make* signal. Each time you release the key, it sends a *break* signal. When processing the **WM_CHAR** message remember that your program is receiving both of these signals. Most of the time you want to take an action only on key press, not on key release. To check for this you must examine the state of the **KC_KEYUP** flag in the first parameter. If it is 0, the key is being pressed; if it is 1, the key is being released.

The following program reads keys from the keyboard and displays normal characters on the screen. It processes the make and skips the break signal. Keep in mind that before the window created by this program can receive input, you must click on the window to make it active. (Only when the window is active does it become the focus of the keyboard.) Notice that this program uses **WinGetPS** and **WinReleasePS**.

```
/* The program reads keystrokes. */

#define INCL_WIN
#define INCL_GPI

#include <os2.h>
#include <stddef.h>  /* get definition of NULL */

void far * pascal far window_func(void far *, unsigned short,
                          void far *, void far *);

char class[] = "MyClass";

main()
{
  void far *hand_ab;
  void far *hand_mq;
  void far *hand_frame, far *hand_client;
  QMSG q_mess;

  hand_ab = WinInitialize(NULL);

  hand_mq = WinCreateMsgQueue(hand_ab, 0);

  if(!WinRegisterClass(hand_ab,    /* anchor block */
                  (char far *) class, /* class name */
                  window_func,    /* address of window function */
                  CS_SIZEREDRAW, /* window style */
                  0))             /* no storage reserved */
     exit(1);

  hand_frame = WinCreateStdWindow(HWND_DESKTOP,
                  WS_VISIBLE | FS_SYSMENU |
                  FS_SIZEBORDER | FS_TITLEBAR |
                  FS_VERTSCROLL| FS_HORZSCROLL |
                  FS_MINMAX,
                  (char far *) class,
                  (char far *) "My Window",
```

```
                    0L,    /* resource modules */
                    NULL,
                    0,
                    &hand_client); /* client handle */

  while(WinGetMsg(hand_ab, &q_mess, NULL, 0, 0))
    WinDispatchMsg(hand_ab, &q_mess);

  WinDestroyWindow(hand_frame);

  WinDestroyMsgQueue(hand_mq);
  WinTerminate(hand_ab);
}

/* window function */
void far * pascal far window_func(void far *handle,
                                  unsigned short mess,
                                  void far *parm1,
                                  void far *parm2)
{
  void far *p_space;
  POINTL coords;
  char ch;

  switch(mess) {
    case WM_ERASEBACKGROUND:
        /* By returning TRUE, the PM automatically erases
           the old window each time the window is resized
           or moved.   Without this, your program must
           manually handle erasing the window when it changes
           size or location.
        */
      return(TRUE);

    case WM_CHAR: /* Process keystrokes here. */
      /* process only keypresses, not key releases */
      if((long) parm1 & KC_KEYUP) break;

      if((long) parm1 & KC_CHAR) {
        p_space = WinGetPS(handle);

        /* use overwrite mode */
        GpiSetBackMix(p_space, BM_OVERPAINT);

        coords.x = 20L;
        coords.y = 20L;

        /* extract the character */
        ch = (char) LOUSHORT (parm2);
        /* display the character */
        GpiCharStringAt(p_space, (POINTL far *) &coords,
                        1L,
                        (char far *) &ch);

        WinReleasePS(p_space);
      }
      break;

    default:
      /* All messages not handled by the window_func,
         must be passed along to the PM for default
         processing.
```

```
      */
      return WinDefWindowProc(handle, mess, parm1, parm2);
  }
  return 0L;
}
```

Keep in mind that the virtual key code and the scan code are two separate pieces of information. The scan code more or less relates to a specific keyboard implementation. However, the virtual key code is completely under the control of OS/2 and the Presentation Mananger, which means that it can map different keys into the virtual codes to accommodate different situations, such as using foreign languages. To see the difference between the virtual and scan codes, substitute this window function in the foregoing program. This version displays the scan and virtual codes for each key pressed.

```
/* window function */
void far * pascal far window_func(void far *handle,
                                  unsigned short mess,
                                  void far *parm1,
                                  void far *parm2)
{
  void far *p_space;
  POINTL coords;
  char ch, str[80];
  int i;

  switch(mess) {
    case WM_ERASEBACKGROUND:
        /* By returning TRUE, the PM automatically erases
           the old window each time the window is resized
           or moved.   Without this, your program must
           manually handle erasing the window when it changes
           size or location.
        */
      return(TRUE);

    case WM_CHAR: /* Process keystrokes here. */
      /* process only keypresses, not key releases */
      if((long) parm1 & KC_KEYUP) break;

      p_space = WinGetPS(handle);

      /* use overwrite mode */
      GpiSetBackMix(p_space, BM_OVERPAINT);
      coords.x = 20L;
      coords.y = 20L;

      /* extract the scan code */
      ch = (char) ((HIUSHORT(parm1) & 0xFF00) >> 8);
      /* display the scan code */
      sprintf(str, "scan code %3d", ch);
      GpiCharStringAt(p_space, (POINTL far *) &coords,
                      (long) strlen(str),
                      (char far *) str);
```

```
      coords.x = 20L;
      coords.y =  L;
      /* extract virtual code */
      i = HIUSHORT(parm2);
      /* display the virtual code */
      sprintf(str, "virtual code %3d", i);
      GpiCharStringAt(p_space, (POINTL far *) &coords,
                      (long) strlen(str),
                      (char far *) str);

      WinReleasePS(p_space);
      break;

    default:
      /* All messages not handled by the window func,
         must be passed along to the PM for default
         processing.
      */
      return WinDefWindowProc(handle, mess, parm1, parm2);
  }
  return 0L;
}
```

A Better Approach to Screen Output

Often the best time for your Presentation Manager-compatible programs to output information to the screen is when a **WM_PAINT** message is received. (Keep in mind that it is not technically wrong to output information to the screen during the processing of other messages, as was done in the previous two examples.) The reason for this is that the Presentation Manager assumes that it is your program's job to maintain and update the screen whenever all or part of the window becomes invalid. A window is invalidated when it is uncovered, resized, or moved. Put another way, when a window's size or position is changed or a previously covered window is uncovered, all or part of the information that was displayed in that window needs to be redrawn. This is the entire purpose of the **WM_PAINT** message. Output performed during the processing of another message is lost if the window is moved or changed (unless, of course, the routine that processes the **WM_PAINT** message can also refresh this output).

To redraw the window each time a **WM_PAINT** message is received the **WM_PAINT** code must be capable of completely reconstructing the screen. To give you a taste of what this entails, the following program rewrites the one that reads a keystroke and displays the key. In this version the code associated with the **WM_CHAR** message simply loads the variable **ch**. It is the code associated with the **WM_PAINT** message that actually outputs the character.

```
/* A Second approach to displaying keystrokes on
   the screen.
*/
#define INCL_WIN
#define INCL_GPI

#include <os2.h>
#include <stddef.h>  /* get definition of NULL */

void far * pascal far window_func(void far *, unsigned short,
                          void far *, void far *);

char class[] = "MyClass";
main()
{
  void far *hand_ab;
  void far *hand_mq;
  void far *hand_frame, far *hand_client;
  QMSG q_mess;

  hand_ab = WinInitialize(NULL);

  hand_mq = WinCreateMsgQueue(hand_ab, 0);

  if(!WinRegisterClass(hand_ab,    /* anchor block */
                  (char far *) class, /* class name */
                  window_func,    /* address of window function */
                  CS_SIZEREDRAW, /* window style */
                  0))             /* no storage reserved */
     exit(1);

  hand_frame = WinCreateStdWindow(HWND_DESKTOP,
                  WS_VISIBLE | FS_SYSMENU |
                  FS_SIZEBORDER | FS_TITLEBAR |
                  FS_VERTSCROLL| FS_HORZSCROLL |
                  FS_MINMAX,
                  (char far *) class,
                  (char far *) "My Window",
                  0L,
                  NULL,
                  0,
                  &hand_client); /* client handle */


  while(WinGetMsg(hand_ab, &q_mess, NULL, 0, 0))
    WinDispatchMsg(hand_ab, &q_mess);

  WinDestroyWindow(hand_frame);

  WinDestroyMsgQueue(hand_mq);
  WinTerminate(hand_ab);
}

/* window function */
void far * pascal far window_func(void far *handle,
                                  unsigned short mess,
                                  void far *parm1,
                                  void far *parm2)
{
  void far *p_space;
  POINTL coords;
  static char ch='\0';
```

```
switch(mess) {
  case WM_PAINT:
      /* Refresh the window each time the WM_PAINT message
         is received.
      */
        p_space = WinGetPS(handle);

        /* use overwrite mode */
        GpiSetBackMix(p_space, BM_OVERPAINT);

        coords.x = 20L;
        coords.y = 20L;

        /* display the character */
        GpiCharStringAt(p_space, (POINTL far *) &coords,
                        1L,
                        (char far *) &ch);

        WinReleasePS(p_space);

      break;

    case WM_ERASEBACKGROUND:
        /* By returning TRUE, the PM automatically erases
           the old window each time the window is resized
           or moved.   Without this, your program must
           manually handle erasing the window when it changes
           size or location.
        */
      return(TRUE);

    case WM_CHAR: /* Process keystrokes here. */
      /* process only keypresses, not key releases */
      if((long) parm1 & KC_KEYUP) break;

      if((long) parm1 & KC_CHAR) {
        ch = (char) LOUSHORT (parm2);
        /* update the window each time a key is pressed */
        WinUpdateWindow(handle);
      }
      break;

    default:
      /* All messages not handled by the window_func,
         must be passed along to the PM for default
         processing.
      */
      return WinDefWindowProc(handle, mess, parm1, parm2);
  }
  return 0L;
}
```

This approach to screen output is very common in Presentation Manager-compatible programs. In this method all output is directed to internal buffers, which are written to the screen when the **WM_PAINT** message is received.

A GRAPHICS EXAMPLE

As you should know, it is very difficult to perform graphics output by using only the core API services. The creators of OS/2 left the task of graphics display to the Presentation Manager. This section shows a short example of graphics output.

The Current Position Approach to Graphics

As discussed earlier in this chapter, the Presentation Manager maintains a pointer to the currently active screen location. The Presentation Manager graphics system uses this current location to streamline many of its graphics services, such as those that draw lines and boxes. To understand how this works, first consider the more traditional approach to the basic graphics functions.

In a traditional graphics system the function that draws a line is defined something like this:

```
drawline(startX, startY, endX, endY)
```

Here the starting and ending points of the line are both specified explicitly in the function parameters. In the traditional method all graphics functions specify both the beginning and ending points of the object to be drawn (where applicable, of course). However, the Presentation Manager uses a fundamentally different approach based on the current position. In this method the call to the line-drawing function specifies only the endpoint of the line. The start of the line is the current position. That is, the line-drawing service found in the Presentation Manager draws a line from the current position to the specified endpoint. The same principle applies to the service that draws a box. You simply call the box-drawing function with the coordinates of the corner opposite the current position, and the box is drawn using the current position and the specified opposite corner.

The reason that the Presentation Manager uses the current position approach is speed. Because each parameter in a call takes time to push onto the stack, the fewer the parameters, the faster the call is executed. The most effective graphics are those that can be displayed very quickly. In many drawing situations the next graphics event begins where the last one left off, making the display of graphic information

very fast. Of course the Presentation Manager contains a service that allows you to set the current position explicitly should the need arise.

The screen coordinates for the graphics subsystem are the same as for the text routines: The lower left corner is 0,0. The maximum *x* and *y* values are determined by the size of the window and, ultimately, by the resolution of the screen.

Drawing Lines and Boxes

The Presentation Manager supplies several graphics functions, but this section explores only three of the most common: **GpiSetPel**, **GpiLine**, and **GpiBox**. These services draw a point, line, and box, respectively. Their prototypes are

```
long GpiSetPel(void far *p_space, POINTL far *loc);
long GpiLine(void far *p_space, POINTL far *loc);
long GpiBox(void far *p_space, long style, POINTL far *loc,
            long horiz_round, long vert_round);
```

where *p_space* is the handle of the presentation space being written to. All functions use the current foreground color to draw the object.

For **GpiSetPel** the structure pointed to by *loc* contains the coordinates of the pel that will be written. The current position is unchanged by this service.

For **GpiLine** the structure pointed to by *loc* contains the endpoint of the desired line. The start of the line is the current position. After the call to **GpiLine**, the current position is set to the end of the line specified by *loc*.

For **GpiBox** the structure pointed to by *loc* is the corner opposite the current position. A rectangle is drawn through these two corners. The value of *style* determines whether the box is outlined, filled, or both. The valid values, along with their macro names as defined by Microsoft are shown here. The current position is unchanged by this service.

Macro Name	Value	Meaning
DRO_FILL	1L	Fill the box.
DRO_OUTLINE	2L	Outline the box.
DRO_OUTLINEFILL	3L	Fill and outline the box.

Outlining and filling are done in the current drawing color.

If any of these functions is called using invalid coordinates, OS/2 returns an error message.

Setting the Current Position

To set the current position explictly use **GpiSetCurrentPosition**, whose prototype is

```
unsigned short GpiSetCurrentPosition(void far *p_space,
                                     POINTL far *loc);
```

Here *p_space* is the handle of the presentation space, and the structure pointed to by *loc* contains the coordinates of the pel to make the current position. If you specify an invalid coordinate, the service returns false.

A Short Graphics Demo Program

The following program demonstrates the graphics services just discussed:

```
/* This program demonstrates some graphics services. */

#define INCL_WIN
#define INCL_GPI

#include <os2.h>
#include <stddef.h>  /* get definition of NULL */

void far * pascal far window_func(void far *, unsigned short,
                                  void far *, void far *);

char class[] = "MyClass";

main()
{
  void far *hand_ab;
  void far *hand_mq;
  void far *hand_frame, far *hand_client;
  QMSG q_mess;

  hand_ab = WinInitialize(NULL);

  hand_mq = WinCreateMsgQueue(hand_ab, 0);

  if(!WinRegisterClass(hand_ab,     /* anchor block */
                       class,       /* class name */
                       window_func, /* address of window function */
                       CS_SIZEREDRAW, /* window style */
                       0))          /* no storage reserved */
     exit(1);
```

```
    hand_frame = WinCreateStdWindow(HWND_DESKTOP,
                    WS_VISIBLE | FS_SYSMENU |
                    FS_SIZEBORDER | FS_TITLEBAR |
                    FS_MINMAX,
                    (char far *) class,
                    (char far *) "My Window",
                    0L,
                    NULL,
                    0,
                    &hand_client);

    while(WinGetMsg(hand_ab, &q_mess, NULL, 0, 0))
      WinDispatchMsg(hand_ab, &q_mess);

    WinDestroyWindow(hand_frame);

    WinDestroyMsgQueue(hand_mq);
    WinTerminate(hand_ab);
}

/* window function */
void far * pascal far window_func(void far *handle,
                                  unsigned short mess,
                                  void far *parm1,
                                  void far *parm2)
{
    void far *p_space;
    POINTL coords;
    char ch;

    switch(mess) {
      case WM_PAINT:
        p_space = WinBeginPaint(handle, NULL, NULL);

        GpiSetBackColor(p_space, CLR_RED);
        GpiSetBackMix(p_space, BM_OVERPAINT);

        /* set current position */
        coords.x = 0L;
        coords.y = 0L;
        GpiSetCurrentPosition(p_space, (POINTL far *) &coords);

        /* draw two lines */
        coords.x = 100L;
        coords.y = 100L;
        GpiLine(p_space, (POINTL far *) &coords);
        coords.x = 200L;
        coords.y = 100L;
        GpiLine(p_space, (POINTL far *) &coords);

        /* draw a filled box */
        coords.x = 300L;
        coords.y = 200L;
        GpiBox(p_space, DRO_FILL, (POINTL far *) &coords, 0L, 0L);

        /* draw a point */
        coords.x = 20L;
        coords.y = 30L;
        GpiSetPel(p_space, (POINTL far *) &coords);

        WinEndPaint(handle);
        break;
```

```
    case WM_ERASEBACKGROUND:
      return(TRUE);

    default:
      return WinDefWindowProc(handle, mess, parm1, parm2);
  }
  return 0L;
}
```

You might find it interesting to play with the various settings or change the drawing color.

A QUICK INTRODUCTION TO MENUS

One of the best features of the Presentation Manager from a programming point of view is the ease with which it integrates menus into a program. Virtually all the work is done for you, including the automatic alignment of the menu items, the integration of the mouse and keyboard into the selection process, and the cancel-selection process. The final section of this chapter describes how you add menus to your Presentation Mangager-compatible programs.

Before you can explore menus, you must understand the concepts that underlie not only them but also other important Presentation Manager tools. Toward this end this section begins with a discussion of resources and the resource compiler.

Resources

One of the most important abstractions supported by the Presentation Manager is the *resource*. The Presentation Manager is capable of managing several resources, including menus, icons, dialog boxes, bit-mapped graphics images, mouse pointers, and string tables. Although this book deals only with menus, OS/2 handles all resources in basically the same way.

Essentially a resource is an object that contains information used by the Presentation Manager. This object is more or less a "black box" as far as your program is concerned because the object is added to your program *after* the program has been compiled (or assembled) and nothing in it can be directly accessed by your program. Instead, the Presentation Manager acts as a link between your program and the resource.

The resources used by your program are defined in a *resource source file* (sometimes called a script file). By convention all resource source files use the .RC extension. The resource source file should have the same file name as the program that uses it. Inside the resource source file you define the resources your program needs. This file recognizes various commands that are used to define resources. (Menu commands will be discussed shortly.) This file is then compiled into a .RES file by the resource compiler. (The resource compiler supplied by Microsoft is called RC.EXE.) The .RES file is added to your program's .EXE file, once again by using the resource compiler. The output of this final step is an .EXE file that contains both your program and its resources. Microsoft's resource compiler allows you to translate the resource source file and add the ouput to your program's .EXE file in one step if desired. Using this approach the compilation sequence is

1. Compile your program.
2. Link your program.
3. Use the resource compiler to add resources to your program.

In practical terms you can use the following batch file to compile, link, and add resources to your programs. It assumes that your program, its definition, and its resource files have the same file name and the conventional extensions.

```
CL -c -G2sw %1.c
LINK %1,,, os2, %1.def;
RC %1
```

Defining Menus in the Resource File

The keyword that signals a menu definition within a resource file is **MENU**. The **MENU** statement takes the general form

```
MENU menu_id {
   SUBMENU "entry1", entry1_id {
      MENUITEM "item1", item1_id
      MENUITEM "item2", item2_id
   }
```

```
    SUBMENU "entry2", entry2_id {
        MENUITEM "item1", item1_id
        MENUITEM "item2", item2_id
    }
    .
    .
    .
    SUBMENU "entryN", entryN_id {
        MENUITEM "item1", item1_id
        MENUITEM "item2", item2_id
    }
}
```

Here the uppercase terms are keywords and the lowercase words are filled in by you.

The entire menu is started with the **MENU** command, and the menu is identified by the value of *menu_id*, which must be an integer. This value is its *resource identifier*, which will be needed by the Presentation Manager. Each menu option is specified by using the **SUBMENU** keyword. The string specified between the quotes will be displayed on the menu bar, which will appear just below the title bar in the window. The number following each **SUBMENU** string is its identifier. Each menu item under a submenu is given a label and associated with an integer using the **MENUITEM** command. Although technically these integers do not have to be unique, most of the time you will want them to be. The numbers associated with the menu items are sent to your window function. To identify a unique selection, the numbers associated with the items must all be different.

For example, the following is an actual resource source file for a simple, two-entry menu:

```
MENU 1 {
  SUBMENU "Advance", 1 {
    MENUITEM "up", 1
    MENUITEM "down", 2
  }
  SUBMENU "Retreat", 2 {
    MENUITEM "up", 3
    MENUITEM "down", 4
  }
}
```

In general the numbers associated with the various menu items should be unique because they are used to identify the item selected by the user.

Adding the Menu to the Window

To display a menu resource in a window you must add it to the window by using the **WinCreateStdWindow** service as follows:

1. Add the **FS_MENU** (4L) to the style (second) parameter. This lets OS/2 know that you will be using a menu resource.
2. Pass the Presentation Manager the identifier of the menu by using the resources parameter. For example, this sample call uses resource number 1, which implies menu number 1:

```
hand_frame = WinCreateStdWindow(HWND_DESKTOP,
                WS_VISIBLE | FS_SYSMENU |
                FS_SIZEBORDER | FS_TITLEBAR |
                FS_VERTSCROLL| FS_HORZSCROLL |
                FS_MINMAX | FS_MENU, /* includes a menu */
                (char far *) class,
                (char far *) "My Window",
                0L,   /* resource modules */
                NULL,
                1,  /* resource identifier */
                &hand_client); /* client handle */
```

When the window is created, a menu bar will appear beneath the title bar. In this example, the menu bar will contain the selections "Advance" and "Retreat."

Keep one thing firmly in mind: A resource file can contain resources for several different windows. However, all the resources for a specific window must use the same resource identifier.

Receiving Menu Messages

Each time you make a menu selection the Presentation Manager passes to your program a **WM_COMMAND** message. The low-order word of the first parameter contains the identifier associated with the item selected. For present purposes, you can ignore the other information passed with the **WM_COMMAND** message.

A Sample Menu

To see a menu in action create this resource file:

```
MENU 1 {
  SUBMENU "Test", 1 {
    MENUITEM "Option 1", 1
    MENUITEM "Option 2", 2
  }
  SUBMENU "Sample", 2 {
    MENUITEM "Option 1", 3
    MENUITEM "Option 2", 4
  }
}
```

Next enter this program:

```
/* A menu example. */

#define INCL_WIN
#define INCL_GPI

#include <os2.h>
#include <stddef.h>  /* get definition of NULL */

void far * pascal far window_func(void far *, unsigned short,
                          void far *, void far *);

char class[] = "MyClass";

main()
{
  void far *hand_ab;
  void far *hand_mq;
  void far *hand_frame, far *hand_client;
  QMSG q_mess;
  hand_ab = WinInitialize(NULL);

  hand_mq = WinCreateMsgQueue(hand_ab, 0);

  if(!WinRegisterClass(hand_ab,    /* anchor block */
                    (char far *) class, /* class name */
                    window_func,    /* address of window function */
                    CS_SIZEREDRAW, /* window style */
                    0))             /* no storage reserved */
     exit(1);

  hand_frame = WinCreateStdWindow(HWND_DESKTOP,
                  WS_VISIBLE | FS_SYSMENU |
                  FS_SIZEBORDER | FS_TITLEBAR |
                  FS_VERTSCROLL| FS_HORZSCROLL |
                  FS_MINMAX | FS_MENU,
                  (char far *) class,
                  (char far *) "My Window",
                  0L,   /* resource modules */
                  NULL,
                  1,
                  &hand_client); /* client handle */
```

```
  while(WinGetMsg(hand_ab, &q_mess, NULL, 0, 0))
    WinDispatchMsg(hand_ab, &q_mess);

  WinDestroyWindow(hand_frame);

  WinDestroyMsgQueue(hand_mq);
  WinTerminate(hand_ab);
}

void far * pascal far window_func(void far *handle,
                                  unsigned short mess,
                                  void far *parm1,
                                  void far *parm2)
{
  void far *p_space;
  POINTL coords;
  static char ch='\0';

  switch(mess) {
      case WM_CREATE:
        /* Perform any necessary initializations here. */

      break;

      case WM_COMMAND:
         p_space = WinGetPS(handle);

         /* use overwrite mode */
         GpiSetBackMix(p_space, BM_OVERPAINT);
        /* see what item selected */
        switch(LOUSHORT(parm1)) {
          case 1:
            coords.x = 20L;
            coords.y = 40L;
            GpiCharStringAt(p_space, (POINTL far *) &coords,
                      8L,
                      (char far *) "test one");
            break;
          case 2:
            coords.x = 110L;
            coords.y = 40L;
            GpiCharStringAt(p_space, (POINTL far *) &coords,
                      8L,
                      (char far *) "test two");
            break;
          case 3:
            coords.x = 20L;
            coords.y = 40L;
            GpiCharStringAt(p_space, (POINTL far *) &coords,
                      10L,
                      (char far *) "sample one");
            break;
          case 4:
            coords.x = 110L;
            coords.y = 40L;
            GpiCharStringAt(p_space, (POINTL far *) &coords,
                      10L,
                      (char far *) "sample two");
            break;

        }
        WinReleasePS(p_space);
        break;
```

```
    case WM_ERASEBACKGROUND:
        /* By returning TRUE, the PM automatically erases
           the old window each time the window is resized
           or moved.   Without this, your program must
           manually handle erasing the window when it changes
           size or location.
        */
      return(TRUE);

    default:
      /* All messages not handled by the window_func,
         must be passed along to the PM for default
         processing.
      */
      return WinDefWindowProc(handle, mess, parm1, parm2);
  }
  return 0L;
}
```

As you can see when you try this program, each menu selection produces a unique response from the program.

CONCLUSION

You have only scratched the surface of the Presentation Manager programming environment. As you can see, writing Presentation Manager-compatible applications is different from creating a traditional program. In many ways it is harder. In the years to come, however, there is little doubt that the graphic interface supported by the Presentation Manager will be pervasive, and whatever effort you expend in learning it now will be returned to you several times in the future.

APPENDIXES

A

80286's MEMORY MODELS

The 80286 supports six different memory models for which a program can be compiled or assembled. A memory model is essentially the conceptual view your program has of memory. Each model treats the memory of the computer differently and governs the size of the code, the data, or both. The model used has a profound effect on your program's speed of execution and the way it accesses the system resources, especially memory.

Throughout this discussion keep one fact firmly in mind: The various memory models are determined solely by the way your program uses the processor's segment registers. They are not different CPU modes.

The six models are called tiny, small, medium, compact, large, and huge. Let's look at how they differ.

TINY MODEL

In a program that uses the tiny model all segment registers are set to the same value and remain more or less fixed throughout the program's lifetime. All addressing is done by using the 16-bit offset. This means that the code, data, and stack must all be within the same 64K segment. This method of compilation produces the smallest, fastest code. The tiny model produces the fastest run times.

SMALL MODEL

In a program compiled for the small model all segment registers are set to values that stay more or less fixed throughout the lifetime of the program. All addressing is done by using the 16-bit offset. However, the code segment is separate from the data, stack, and extra segments, which are in their own segment. This means that the total size of a program compiled this way is 128K split between code and data. The addressing time is the same as for the tiny model, but the program can be twice as big. Many of your programs will be of this model. The small model produces run times as fast as those of the tiny model.

MEDIUM MODEL

The medium model is for large programs where the code exceeds the one-segment restriction of the small model. Here the code can use multiple segments and requires 32-bit pointers, but the code, data, and extra segments are in their own segment and use 16-bit addresses. This is good for large programs that use little data. Your programs will run more slowly as far as function calls are concerned, but references to data will be as fast as in the small model.

COMPACT MODEL

The complement of the medium model is the compact model. In this version program code is restricted to one segment but data can occupy several segments. This means that all accesses to data require 32-bit addresses but the code uses 16-bit addresses. This is good for programs that require large amounts of data but little code. Such a program will run as fast as the small model except when referencing data, which will be slower.

LARGE MODEL

The large model allows both code and data to use multiple segments. However, the largest single item of data, such as an array, is limited to 64K. Use this model when you have large code and data requirements. It is slower than any of the preceding versions.

HUGE MODEL

The huge model is the same as the large model with the exception that individual data items may exceed 64K. This further degrades run time.

OVERRIDING A MEMORY MODEL IN C

During the foregoing discussion you may have been thinking how unfortunate it is that even a single reference to data in another segment would require you to use the compact rather than small model, thus slowing the execution of the entire program even though only an isolated part of it actually needs a 32-bit pointer. In general this sort of situation can present itself in a variety of ways. For example, it is necessary to use 32-bit addressing to access an API service routine. The solution to this and other related problems is the *segment override* type modifiers, which are enhancements provided with most 80286-based C compilers. They are

near far

These modifiers can be applied only to pointers or functions. When they are applied to pointers, they affect the way data is accessed. When applied to functions, they affect the way you call and return from the function.

These modifiers follow the base type and precede the variable name. For example, this declares a **far** pointer called **f_pointer**:

```
char far *f_pointer;
```

Let's look at these modifiers now.

far

By far the most common model override is the **far** pointer. It is very common to want to access some region of memory that is (or may be) outside the program's data segment, such as a data segment returned by an API service. However, if the program is compiled for one of the

large data models, all accesses to data—not just the one outside the data segment—become very slow. The solution to this problem is explicitly to declare **far** pointers to the memory that is outside the current data segment. In this way only references to objects actually far away will incur the additional overhead.

The use of **far** as a function modifier allows a small model program to call routines outside its code segment, such as API services. In such cases the use of **far** ensures that the proper calling and returning sequences are used.

near

A **near** pointer is a 16-bit offset that uses the value of the appropriate segment to determine the actual memory location. The **near** modifier forces C to treat the pointer as a 16-bit offset to the segment contained in the **DS** register. You use a **near** pointer when you have compiled a program using either the medium, large, or huge memory model and wish to reference data within the program's data segment.

Using **near** on a function causes that function to be treated as if it were compiled using the small code model. When a function is compiled with either the tiny, small, or compact model, all calls to the function place a 16-bit return address on the stack. If a function is compiled with the large code model, a 32-bit address is pushed onto the stack. Therefore, in programs that are compiled for the large code model, a highly recursive function should be declared as **near** to conserve stack space and speed execution time.

B

FUNCTION PROTOTYPES

In C a function that returns a value other than **int** must be declared prior to its use so that the compiler can generate the proper return codes. In ANSI standard C you can take this idea one step further by also declaring the number and types of the function's arguments. This expanded definition is called a *function prototype*. Function prototypes are not part of the original UNIX C but were added by the ANSI standardization committee. They enable C to provide stronger type checking, somewhat similar to that provided by languages such as Pascal. Function prototypes also provide a convenient means of documenting the calling syntax of a function.

In a strongly typed language the compiler issues error messages if functions are called with arguments that cause illegal type conversions or with a different number of arguments. Although C is designed to be very forgiving, some type conversions are simply not allowed. For example, it is an error to attempt to convert a pointer into a **float**. Using function prototypes will catch and prevent this sort of error.

A function prototype takes the general form

```
type function_name(arg_type1, arg_type2,...,arg_typeN);
```

where *type* is the type of value returned by the function and *arg_type* is the type of each argument.

For example, this program will produce an error message because there is an attempt to call **func()** with a pointer instead of the **float** required:

```
/* This program uses function prototypes to
   enforce strong type checking in the calls
   to func().

   The program will not compile because of the
   mismatch between the type of the arguments
   specified in the function's prototype and
   the type of arguments used to call the function.

*/

float func(int, float); /* prototype */

main()
{
  int x, *y;

  x = 10;  y = 10;
  func(x, y);  /* type mismatch */
}

float func(int x, float y)
{
  printf("%f", y/(float)x);
}
```

Not only does the use of function prototypes help you trap bugs before they occur, but also they help verify that your program is working correctly by not allowing functions to be called with mismatched arguments or an incorrect number of arguments. It is generally a good idea to use prototyping in larger programs or in situations in which several programmers are working on the same project.

CLASSIC VERSUS MODERN PARAMETER DECLARATIONS

It is possible to declare parameters to a function in two different ways: the traditional (sometimes called classic) or the modern. The traditional method is used by the earlier C compilers, while the modern form is defined by the ANSI standard. Let's look at both.

In the traditional form only a function parameter's names are placed between the parentheses following the function's name. Before the function's opening curly brace, the parameters are declared using a

syntax identical to the variable declaration. For example, this code declares a function with two variables, **a** and **b**, of types integer and real, respectively.

```
int f1(a, b)
int a;
float b;
{
 .
 .
 .
```

Although there is nothing formally wrong with the traditional method, the newer ANSI standard offers an alternative approach based on the prototype syntax.

In the modern approach both the type and the name of the variable are enclosed in parentheses and placed in the argument list that follows the function's name. That is, the function parameter declaration takes a similar form to the prototype declaration except that the name of the parameter must be included. The modern declaration method takes the general form

```
type function_name(type parm1, type parm2,..., type parmN)
{
    body of function
}
```

where *type* is the type of the parameter that follows and *parm* is the name of the parameter.

For example, the function **func( )** from the prototype example of the previous section is written like this using the modern parameter declaration method:

```
float f1(int a, float b)
{
 .
 .
 .
```

There is a very fine technical difference between the ways the compiler handles each form, but for most situations the difference is academic.

C

A REVIEW OF C

This appendix aids the inexperienced C programmer by clarifying aspects of the language. It is a reference guide and not a tutorial.

THE ORIGINS OF C

The C language was invented and first implemented by Dennis Ritchie on a DEC PDP-11 using the UNIX operating system. C is the result of a process that started with Martin Richards' development of BCPL, which is still used primarily in Europe. BCPL prompted Ken Thompson to invent a language called B, which led to the development of C.

For many years, the de facto standard for C was the one supplied with the UNIX Version 5 operating system and described in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Englewood Cliffs, N.J.: Prentice-Hall, 1978). As the popularity of microcomputers increased, a great number of C implementations were created. Most of these implementations were highly compatible with each other on the source-code level. However, because no standard existed, there were some discrepancies.

To correct this situation, a committee established in the summer of 1983 began work on the creation of an ANSI standard that would finally define the C language. As of this writing, the proposed standard is almost complete and its adoption by ANSI is expected soon.

C AS A STRUCTURED LANGUAGE

C is commonly considered to be a structured language with some similarities to ALGOL and Pascal. Although the term *block-structured language* does not strictly apply to C in an academic sense, C is informally part of that language group. The distinguishing feature of a block-structured language is *compartmentalization of code and data.* This means the language can separate and hide from the rest of the program all information and instructions necessary to perform a specific task. Compartmentalization is generally achieved by subroutines with local variables, which are temporary. This makes it possible to write subroutines so that the events occurring in them have no effect on other parts of the program. Excessive use of global variables (variables known throughout the entire program) may allow bugs to creep into a program by allowing unwanted side effects. In C all subroutines are discrete functions.

Functions are the building blocks of C in which all program activity occurs. They allow specific tasks in a program to be defined and coded separately. After debugging a function that uses only local variables, you can rely on the function to work properly in various situations without creating side effects in other parts of the program. All variables declared in that particular function will be known only to that function.

Using blocks of code also creates program structure in C. A *block of code* is a logically connected group of program statements that can be treated as a unit. It is created by placing lines of code between opening and closing curly braces, as shown here:

```
if(x<10) {
  printf("Invalid input - retry");
  done = 0;
}
```

In this example, the two statements after the **if** (between curly braces) are both executed if **x** is less than 10. These two statements and the braces represent a block of code. They are linked together: one of the statements cannot execute without the other also executing. In C every statement can be either a single statement or a block of statements. The use of code blocks creates readable programs with logic that is easy to follow.

C is a programmer's language. Unlike most high-level computer languages, C imposes few restrictions on what you can do with it. By using C a programmer can avoid using assembly code for all but the most demanding situations. In fact one motive for the inventing of C was to provide an alternative to assembly language programming.

Assembly language uses a symbolic representation of the actual binary code that the computer directly executes. Each assembly language operation is a single operation for the computer to perform. Although assembly language gives programmers the potential for accomplishing tasks with maximum flexibility and efficiency, it is notoriously difficult to work with when developing and debugging a program. Furthermore, since assembly language is unstructured by its nature, the final program tends to be "spaghetti code"—a tangle of jumps, calls, and indexes. This makes assembly language programs difficult to read, enhance, and maintain.

C was initially used for systems programming. A *systems program* is part of a large class of programs that form a portion of the operating system of the computer or its support utilities. For example, the following are commonly called systems programs:

- Operating systems
- Interpreters
- Editors
- Assemblers
- Compilers
- Data base managers

As C grew in popularity, many programmers began to use C to program all tasks because of its portability and efficiency. Since there are C compilers for virtually all computers, it is easy to compile and run code written for one machine on another machine with few or no changes. This portability saves both time and money. C compilers also tend to produce tight, fast object code—faster and smaller than most BASIC compilers, for example.

Perhaps the real reason that C is used in all types of programming tasks is because programmers like it. C has the speed of assembler and the extensibility of FORTH, with few of the restrictions of Pascal. A C programmer can create and maintain a unique library of functions that

have been tailored to his or her own personality. Because C allows—indeed encourages—separate compilation, large projects are easy to manage.

A REVIEW OF C

As defined by the proposed ANSI standard, the 32 keywords shown in Table C-1, combined with the formal C syntax, form the C programming language.

In addition to these keywords, several compilers designed for use on the 8086 family of processors or multilanguage programming environments have added the following to allow greater control over the way memory and other system resources are used:

_cs	_ds	_es	_ss
cdecl	far	huge	interrupt
near	pascal		

All C keywords are in lowercase letters. Uppercase or lowercase makes a difference in C; that is, **else** is a keyword, **ELSE** is not.

VARIABLES—TYPES AND DECLARATION

C has five built-in data types, as shown in Table C-2. With the exception of **void**, all these data types can be modified through the use of the C type modifiers:

signed
unsigned
short
long

Variable names are strings of letters from 1 to 32 characters in length. The ANSI standard states that at least six characters will be significant. For clarity the underscore may also be used as part of the variable name (for example, **first_time**). Remember that in C uppercase and lowercase are different—**test** and **TEST** are two different variables.

Table C-1. List of Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table C-2. Data Types and C Keyword Equivalents

Data Type	C Keyword Equivalent
character	char
integer	int
floating point	float
double floating point	double
value-less	void

All variables must be declared prior to use. The general form of the declaration is

type variable_name;

For example, to declare **x** to be a float, **y** to be an integer, and **ch** to be a character, type

```
float x;
int y;
char ch;
```

In addition to the built-in types, you can create combinations of built-in types by using **struct** and **union**. You can also create new names for variable types by using **typedef**.

A *structure* is a collection of variables grouped and referenced under one name. The general form of a structure declaration is

```
struct struct_name {
  element 1;
  element 2;
  .
  .
  .
  element N;
} struct_variable;
```

For example, the following structure has two elements: **name**, a character array, and **balance**, a floating-point number:

```
struct client {
  char name[80];
  float balance;
};
```

Use the dot operator to reference individual structure elements if the structure is global or declared in the function referencing it. Use the arrow operator in all other cases.

Two or more variables sharing the same memory define a **union**. The general form for a **union** is

```
union union_name {
  element 1;
  element 2;
  .
  .
  .
  element N;
} union_variable;
```

The elements of a **union** overlie each other. For example, the following declares a **union t** that looks like Figure C-1 in memory:

```
union tom {
  char ch;
  int x;
} t;
```

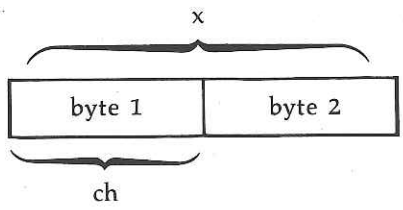


Figure C-1. The **union t** in memory

Reference the individual variables that comprise the **union** by using the dot operator. Use the arrow operator with pointers to unions.

Another type of variable that can be created, called an *enumeration*, is a list of objects or values (depending on how you interpret it). An *enumeration type* is a specification of the list of objects that belong to the enumeration. When you declare a variable to be of an enumeration type, its values can be only those defined by the enumeration.

To create an enumeration, use the keyword **enum**. For example, the following short program defines an enumeration of cities called **cities**, and the variable **c** of type **cities**. Finally, the program assigns **c** the value "Houston."

```
enum cities {Houston, Austin, Amarillo };
enum cities c;

main()
{
  c=Houston;
}
```

The general form of an enumeration type is

enum name { list of values };

The Storage-Class Type Modifiers

Use the type modifiers **extern**, **auto**, **register**, **const**, **volatile**, and **static** to alter the way C creates storage for the variables that follow.

If you place the **extern** modifier before a variable name, the compiler knows that the variable has been declared elsewhere. The **extern** modifier is most commonly used when two or more files share the same global variables.

An **auto** variable is created on entry into a block and is destroyed on exit. For example, all variables defined inside a function are **auto** by default. **Auto** variables can be valuable in specialized or dedicated systems where RAM is in short supply.

You can use the **register** modifier only on local integer or character variables. This modifier causes the compiler to attempt to keep that value in a register of the CPU instead of placing it in memory, which makes all references to that variable extremely fast. For example, the following function uses a **register** loop control:

```
f1()
{
  register int t;

  for(t=0;t<10000;++t) {
    .
    .
    .
  }
}
```

Variables of type **const** cannot be changed during your program's execution. The compiler is free to place variables of this type into ROM. For example, the following line

```
const int a;
```

creates an integer called **a** that cannot be modified by your program, but can be used in other types of expressions. A **const** variable receives its value either from an explicit initialization or by some hardware-dependent means. The inclusion of *const* type variables aids in the development of applications for ROM.

The **volatile** modifier tells the compiler that a variable's value can be changed in ways not explicitly specified by the program. For example, a global variable's address can be passed to the clock routine of the operating system and used to hold the real time of the system. In this situation the contents of the variable are altered without any explicit

assignment statements in the program. To achieve higher performance some C compilers automatically optimize certain expressions by assuming that the contents of a variable are unchanged inside that expression. The **volatile** modifier prevents this optimization in the rare instances where this is not true.

You can add the **static** modifier to any of the previously mentioned variables. The **static** modifier instructs the C compiler to keep a local variable in existence during the lifetime of the program instead of creating and destroying it. Remember that the values of local variables are discarded when a function finishes and returns. By using **static** you can maintain their values between function calls.

Addressing Type Modifiers

Several C compilers designed for the 8086 processor family have added the following modifiers that can be applied to pointers so that you can explicitly control, and override, the default addressing mode used to compile your program:

_cs	_ds	_es	_ss
far	near	huge	

Arrays

You can declare arrays on any of the previously mentioned data types. For example, to declare an integer array **x** of 100 elements, write

```
int x[100];
```

This creates an array that is 100 elements long; the first element is 0 and the last is 99. For example, the following loop loads the numbers 0 through 99 into array **x**:

```
for(t=0;t<100; t++) x[t]=t;
```

You declare multidimensional arrays by placing the additional dimensions inside additional brackets. For example, to declare a 10 x 20 integer array, you write

```
int x[10][20];
```

OPERATORS

C has a rich set of operators that can be divided into the following classes: *arithmetic, relational and logical, bitwise, pointer, assignment,* and *miscellaneous.*

Arithmetic Operators

C has the seven arithmetic operators shown in Table C-3.

The precedence of these operators is

Highest	++ −− −(unary minus)
	* / %
Lowest	+ −

Operators on the same precedence level are evaluated left to right.

Relational and Logical Operators

Relational and logical operators are used to produce TRUE/FALSE results and are often used together. In C any nonzero number evaluates TRUE; however, a C relational or logical expression produces 1 for TRUE and 0 for FALSE. Table C-4 shows the relational and logical operators.

The precedence of these operators is

Highest	!
	>>=<<=
	== !=
	&&
Lowest	||

For example, the following expression evaluates TRUE:

```
(100<200) && 10
```

Table C-3. Arithmetic Operators

Operator	Action
−	Subtraction, unary minus
+	Addition
*	Multiplication
/	Division
%	Modulo division
−−	Decrement
++	Increment

Table C-4. Relational and Logical Operators

Relational Operators

Operator	Meaning
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal

Logical Operators

Operator	Meaning
&&	AND
||	OR
!	NOT

Bitwise Operators

Unlike most other programming languages, C provides bitwise operators that manipulate the actual bits inside a variable. The bitwise operators can be used only on integers or characters. They are shown in Table C-5.

Table C-5. Bitwise Operators

Operator	Meaning
&	AND
|	OR
^	XOR
~	One's complement
>>	Right shift
<<	Left shift

The truth tables for AND, OR, and XOR are

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

These rules are applied to each bit in a byte when the bitwise AND, OR, and XOR operations are performed. For example,

```
  0100    1101
& 0011    1011
-------------------
  0000    1001

  0100    1101
| 0011    1011
-------------------
  0111    1111

  0100    1101
^ 0011    1011
-------------------
  0111    0110
```

In a program, you use the &, |, and ^ like any other operators, as shown here:

```
main()
{
  char x,y,z;

  x = 1; y = 2; z = 4;

  x = x & y;  /* x now equals zero */

  y = x | z;  /* y now equals 4 */
}
```

The one's complement operator (~) inverts all the bits in a byte. For example, if the character variable **ch** has the bit pattern

```
0 0 1 1  1 0 0 1
```

then

```
ch=~ch;
```

places the bit pattern

```
1 1 0 0  0 1 1 0
```

into **ch**.

The right shift and left shift operators move all bits in a byte or a word right or left by some specified number of bits. As bits are shifted, 0s are brought in. The number on the right side of the shift operator specifies the number of positions to shift. The general forms of the shift operators are

variable >> number of bit positions
variable << number of bit positions

For example, given the bit pattern

```
0 0 1 1  1 1 0 1
```

a shift right yields

```
0 0 0 1  1 1 1 0
```

while a single shift left produces

```
0 1 1 1  1 0 1 0
```

A shift right is effectively a division by 2 and a shift left is a multiplication by 2. The following code fragment first multiplies and then divides the value in x by 2:

```
int x;
x=10;
x=x<<1;
x=x>>1;
```

Because of the way negative numbers are represented inside the machine, you must be careful when you try to use a shift for multiplication or division. Moving a 1 into the most significant bit position makes the computer think that the number is a negative number.

Note: You use the bitwise operators to modify the value of a variable. They differ from the logical and relational operators, which produce a TRUE or FALSE result.

The precedence of the bitwise operators is

Highest	~
	>> <<
	&
	^
Lowest	|

Pointer Operators

Pointer operators are important in C: They not only allow strings and arrays to be passed to functions, but also allow C functions to modify their calling arguments. The two pointer operators are & and *. (Unfortunately, these operators are the same as the bitwise AND and multiply symbols, which are completely unrelated to them.)

The & operator returns the address of the variable it precedes. For example, if the integer **x** is located at memory address 1000,

```
y = &x;
```

places the value 1000 into **y**. The & can be read as "the address of." For example, the previous statement could be read as "Place the address of **x** into **y**."

The * operator uses the value of the variable it precedes as the address of the information in memory. For example,

```
y = &x;
*y = 100;
```

places the value 100 into **x**. The * can be read as "at address." In this example it could be read as, "Place the value 100 at address **y**." You can also use the * operator on the right-hand side of an assignment. For example,

```
y = &x;
*y = 100;
z = *y/10;
```

places the value of 10 into **z**.

These operators are called pointer operators because they are designed to work on *pointer variables.* A pointer variable holds the address of another variable; in essence, it "points" to that variable as shown in Figure C-2.

Pointers of Type void

A pointer of type **void** is a generic pointer and can point to any type of object. This implies that you can assign a pointer of any type to pointers of type **void** (and vice versa) if you use the appropriate type casts. To declare a **void** pointer you use a declaration similar to the following:

```
void *p;
```

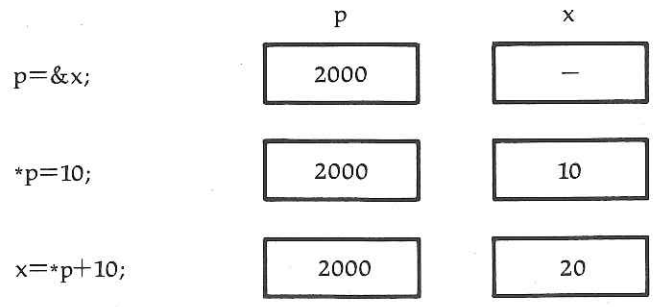


Figure C-2. Pointer operations for character pointer **p** and integer **x**, with **x** at memory location 2000

The **void** pointer is particularly useful when manipulating various types of pointers with a single routine.

Assignment Operators

The assignment operator in C is the single equal sign. However, C allows a convenient "shorthand" for assignments of the general type

variable1 = variable1 operator expression;

For example:

```
x = x+10;
y = y/z;
```

Assignments of this type can be shortened to the general form

variable1 operator = expression;

In the case of the two examples, they can be shortened to

```
x += 10;
y /= z;
```

Experienced C programmers often use the shorthand notation, so you should become used to it.

The ? Operator

The **?** operator is a ternary operator (that is, it takes three operands). It is used to replace **if** statements of the general type

```
if expression1 then x=expression2
  else x=expression3
```

The general form of the **?** operator is

```
variable = expression1 ? expression2 : expression3;
```

If *expression1* is TRUE, the value of *expression2* is assigned to *variable;* otherwise, *variable* is assigned the value of *expression3*. For example,

```
x = (y<10) ? 20 : 40;
```

assigns to **x** the value of 20 if **y** is less than 10 or the value of 40 if **y** is not.

The **?** operator exists because a C compiler can produce very efficient code for this statement—much faster than the equivalent **if/else** statement.

Miscellaneous Operators

The **.** (dot) and → (arrow) operators reference individual elements of structures and unions. Use the dot operator on the structure or union itself. Use the arrow operator when only a pointer to a structure or a union is available. For example, consider the following global structure:

```
struct date_time {
  char date[16];
  int time;
} tm;
```

To assign the value "3/12/88" to element **date** of structure **tm**, write

```
strcpy(tm.date, "3/12/88");
```

You use the , (comma) operator mostly in the **for** statement. It causes a sequence of operations to be performed. When you use it on the right side of an assignment statement, the value of the entire expression is the value of the last expression of the comma-separated list. For example, consider the following:

```
y=10;
x = (y=y-5,25/y);
```

After execution, **x** has the value 5 because the original value of **y** (10) is reduced by 5, and then 25 is divided by that value, yielding a result of 5.

Although **sizeof** is also considered a keyword, it is a compile-time operator used to determine the size of a data type in bytes, including user-defined structures and unions. For example,

```
int x;
printf("%d", sizeof(x));
```

prints the number 2 for many compilers.

Parentheses are operators that increase the precedence of the operations inside them. Square brackets perform array indexing.

A *cast* is a special operator that forces the conversion of one data type into another. The general form is

(type) variable

For example, to use the integer **count** in a call to **sqrt()**, which is the square root routine in C's standard library and requires a floating-point parameter, a cast forces **count** to be treated as type **float**:

```
float y;
int count;
count = 10;
y = sqrt((float)count);
```

Figure C-3 lists the precedence of all C operators. Note that all operators—except the unary operators and ?—associate from left to right. The unary operators (*, &, −, and ?) associate from right to left.

FUNCTIONS

A C program is a collection of one or more user-defined functions. One of the functions must be **main()** because execution begins at this function. Historically, **main()** is the first function in a program; however, it could go anywhere.

The general form of a C function is

```
type function_name(parameter list)
{
   body of function
}
```

```
Highest () →.
        ! ~ ++ -- - (type) * & sizeof
        * / %
        + -
        << >>
        <<= >>=
        == !=
        &
        ^
        |
        &&
        ||
        ?:
        = += -= *= /= %= >>= <<= &= ^= |=
Lowest ,
```

Figure C-3. Precedence of C operators

If the function has no parameters, no parameter declaration is needed. The type declaration is optional. If no explicit type declaration is present, the function defaults to an integer. All functions terminate and return to the calling procedure automatically when the last brace is encountered. You can force a return before that by using the **return** statement.

All functions—except those declared as **void**—return a value. The type of the return value must match the type declaration of the function. If no explicit type declaration has been made, the return value is defaulted to integer. If a **return** statement is part of the function, the value of the function is the value in the **return** statement. If no **return** is present, the function returns an unknown value. For example,

```
f1()
{
  int x;

  x = 100;
  return(x/10);
}
```

returns the value 10, whereas

```
f2()
{
  int x;

  x = 100;
  x = x/10;
}
```

returns a random value because no explicit **return** statement is encountered.

If a function is going to return a value other than an integer, its type must reflect this fact. It is also necessary to declare the function prior to any reference to it by another piece of code. This can best be accomplished by making a function declaration in the global definition area of the program. The following example shows how the function **fn()** is declared to return a floating-point value:

```
float fn();

main()
{
  .
  .
  .
```

```
  printf("%f", fn());
  .
  .
  .
}

float fn()
{
  return 12.23;
}
```

Because all functions, except those declared as **void**, have values, they can be used in any arithmetic statement. For example, beginning C programmers tend to write code like this:

```
x = sqrt(y);

z = sin(x);
```

whereas a more experienced programmer would write:

```
z = sin(sqrt(y));
```

Remember that the program must be executed to determine the value of a function. This means that the following code reads keystrokes from the keyboard until a U is pressed:

```
while((ch=getche())!='u') ;
```

This code works because **getche()** must be executed to determine its value, which is the character typed at the keyboard.

The Scope and Lifetime of Variables

C has two general classes of variables: global and local. A global variable is available for use by all functions in the program, while a local variable is known and used only by the function in which it was declared. In some C literature global variables are called *external variables* and local variables are called *dynamic* or *automatic variables*. This appendix uses the terms *global* and *local* because they are more commonplace.

A global variable must be declared outside all functions, including the **main()** function. Global variables are usually placed at the top of

the file before **main()**, because this makes the program easier to read and because a variable must be declared before it is used. A local variable is declared inside a function after the function's opening brace. For example, the following program declares one global variable, **x**, and two local variables, **x** and **y**:

```
int x;

main()
{
  int y;

  y = get_value();
  x = 100;
  printf("%d %d", x, x*y);
}

f1()
{
  int x;

  scanf("%d", &x);
  return x;
}
```

This program multiplies the number entered from the keyboard by 100. Note that the local variable **x** in **f1()** has no relationship to the global variable **x**, because local variables that have the same name as global variables always take precedence over the global ones.

Global variables exist during the entire program. Local variables are created when the function is entered and are destroyed when the function is exited. This means that local variables do not keep their values between function calls. However, you can use the **static** modifier to preserve values between calls.

The formal parameters to a function are also local variables, and, except for receiving the value of the calling arguments, they behave and can be used like any other local variable.

The main() Function

All C programs must have a **main()** function. When execution begins, **main()** is the first function called. You must not have more than one function called **main()**. When **main()** terminates, the program is over and control passes back to the operating system.

The only parameters that **main()** is allowed to have are **argc** and **argv**. The variable **argc** holds the number of command line arguments. The variable **argv** holds a character pointer to those arguments. *Command line arguments* are the information that you type in after the program name when you execute a program. For example, when you compile a C program, you type something like

```
CC MYPROG.C
```

where MYPROG.C is the name of the program you wish to compile.

The value of **argc** is always at least 1, because C considers the program name to be the first argument. The variable **argv** must be declared as an array of character pointers. This is shown in the following short program, which prints your name on the screen.

```
main(argc, argv)
int argc;
char *argv[];
{
     if(argc<2)
       printf("enter your name on the command line.\n");
     else
       printf("hello %s\n",argv[1]);
}
```

Notice that **argv** is declared as a character pointer array of unknown size. The C compiler automatically determines the size of the array necessary to handle all the command line arguments.

Command line arguments give your programs a professional look and feel, and allow you to place them in a batch file for automatic use.

STATEMENT SUMMARY

This section is a brief synopsis of the keywords in C.

auto

The **auto** keyword creates temporary variables upon entry into a block and destroys them upon exit. For example, in

```
main()
{
  for(;;) {
    if(getche()=='a') {
      auto int t;
      for(t=0; t<'a'; t++)
        printf("%d ", t);
    }
  }
}
```

the variable **t** is created only if you press A. Outside the if block, **t** is completely unknown and any reference to it generates a compile-time syntax error.

break

You use the **break** keyword to exit from a **do**, **for**, or **while** loop, bypassing the normal loop condition. You also use it to exit from a **switch** statement.

The following is an example of **break** in a loop:

```
while(x<100) {
  x = get_new_x();
  if(keystroke()) break;  /* key hit on
                             keyboard */
  process(x);

}
```

In this example, if a key is pressed, the loop terminates no matter what the value of **x** is.

A **break** always terminates the innermost **for**, **do**, **while**, or **switch** statement, regardless of the way they are nested. In a **switch** statement, **break** effectively keeps program execution from "falling through" to the next **case**. (Refer to the discussion of **switch** for details.)

case

Refer to the discussion of **switch**.

cdecl

The **cdecl** keyword is not part of the ANSI standard. It forces C to compile a function so that its parameter passing conforms with the

standard C calling convention. You use **cdecl** only when compiling an entire file while using the Pascal option and when you want a specific function to be compatible with C.

const

The **const** modifier tells the compiler that the following variable cannot be modified.

char

The **char** data type declares character variables. For example, to declare **ch** to be character type, write

```
char ch;
```

continue

You use the **continue** keyword to bypass portions of code in a loop and force the conditional test to be performed. For example, the following **while** loop simply reads characters from the keyboard until S is pressed:

```
while(ch=getche()) {
  if(ch!='s') continue;  /* read another char */
  process(ch);
}
```

The call to **process()** will not occur until **ch** contains the character *S*.

default

You use the **default** keyword in the **switch** statement to signal a default block of code to be executed if no matches are found in the **switch**. (See the discussion of **switch**.)

do

The **do** loop is one of three loop constructs available in C. The general form of the **do** loop is

```
do {
   statement block
} while(condition);
```

If only one statement is in the statement block, the braces are not necessary, but they do add clarity to the statement.

The **do** loop is the only loop in C that always has at least one iteration, because the condition is tested at the bottom of the loop.

The **do** loop is commonly used to read disk files. The following code reads a file until an EOF is encountered:

```
do {
  ch=getc(fp);
  store(ch);
} while(!feof(fp));
```

double

The **double** data-type specifier declares double-precision floating-point variables. To declare **d** to be of type **double**, write

```
double d;
```

else

See the discussion of **if**.

enum

The **enum** type specifier creates enumeration types. An enumeration is simply a list of objects, and an enumeration type specifies what that list of objects is. An enumeration type variable can only be assigned values that are part of the enumeration list. For example, the following code declares an enumeration called **color**, declares a variable of that type called **c**, and performs an assignment and a condition test:

```
enum color {red, green, yellow};
enum color c;

main()
{
```

```
  c=red;
  if(c==red) printf("is red\n");
}
```

extern

The **extern** data-type modifier tells the compiler that a variable is declared elsewhere in the program. This modifier is often used in conjunction with separately compiled files that share the same global data and are linked together. In essence **extern** notifies the compiler of a variable without redeclaring it.

For example, if **first** were declared in another file as an integer, in subsequent files you would use the following declaration:

```
extern int first;
```

float

The **float** data-type specifier declares floating-point variables. To declare f to be of type **float**, write

```
float f;
```

for

The **for** loop allows automatic initialization and incrementing of a counter variable. The general form is

```
for(initialization; condition; increment) {
  statement block
}
```

If the *statement block* is only one statement, the braces are not necessary.

Although the **for** allows a number of variations, generally the *initialization* is used to set a counter variable to its starting value. The *condition* is generally a relational statement that checks the counter variable against a termination value, and *increment* increments (or decrements) the counter value.

The following code prints the message "hello" ten times:

```
for(t=0; t<10; t++) printf("hello\n");
```

The next example waits for a keystroke after printing "hello":

```
for(t=0; t<10; t++) {
  printf("hello\n");
  getche();
}
```

goto

The **goto** keyword causes program execution to jump to the label specified in the **goto** statement. The general form of **goto** is

```
goto label;
.
.
.
label:
```

All labels must end in a colon and must not conflict with keywords or function names. Furthermore, a **goto** can branch only within the current function, not from one function to another.

The following example prints the message "right," but not the message "wrong":

```
goto lab1;
  printf("wrong");
lab1:
  printf("right");
```

if

The general form of the **if** statement is

```
if(condition) {
  statement block 1
}
else {
  statement block 2
}
```

If single statements are used, the braces are not necessary. The **else** is optional.

The condition can be any expression. If that expression evaluates to any value other than 0, *statement block 1* executes; otherwise, *statement block 2* executes.

The following code fragment can be used for keyboard input and to look for a *q*, which signifies "quit."

```
ch=getche();
if(ch=='q') {
  printf("program terminated");
  exit(0);
}
else  proceed();
```

int

The **int** type specifier declares integer variables. For example, to declare **count** as an integer, write

```
int count;
```

interrupt

The **interrupt** type specifier is not part of the ANSI standard. It declares functions that are used as interrupt service routines.

long

The **long** data-type modifier declares double-length integer variables. For example, to declare **count** as a long integer, write

```
long int count;
```

pascal

The **pascal** keyword is not defined by the ANSI standard. It forces C to compile a function so that its parameter-passing convention is compatible with Pascal rather than C.

register

The **register** declaration modifier forces an integer or character to be stored in a register of the CPU instead of being placed in memory. It can be used only on local variables. To declare **i** as a register integer, write

```
register int i;
```

return

The **return** keyword forces a return from a function and can be used to transfer a value back to the calling routine.

For example, the following function returns the product of its two integer arguments:

```
mul (int a, int b)
{
   return(a*b);
}
```

Remember that as soon as a **return** is encountered, the function returns and skips any other code in the function.

sizeof

The **sizeof** keyword is a compile-time operator that returns the length of the variable it precedes. For example, the following prints "2" on most computers:

```
printf("%d", sizeof(int));
```

The principal use of **sizeof** is in generating portable code when that code depends on the size of the C built-in data types.

signed

The **signed** type modifier produces a **signed** data type.

short

The **short** data-type modifier declares 1-byte integers. For example, to declare **sh** as a short integer, write

```
short int sh;
```

static

The **static** data-type modifier instructs the compiler to create permanent storage for the local variable that it precedes. This enables the specified variable to maintain its value between function calls. For example, to declare **last_time** as a **static** integer, write

```
static int last_time;
```

struct

The **struct** keyword creates complex or conglomerate variables (called *structures*) that are made up of one or more elements of the seven basic data types. The general form of a structure is

```
struct struct_name {
  type element1;
  type element2;
  .
  .
  .
  type elementn;
} structure_variable_name;
```

You reference the individual elements by using the dot or arrow operator.

switch

The **switch** statement is C's multiway branch statement. It is used to route execution one of several different ways. The general form of the statement is

```
switch(variable) {
   case (constant1): statement set 1;
                     break;
   case (constant2): statement set 2;
                     break;
   .
   .
   .
   case (constant n): statement set N;
                      break;
   default: default statements;
}
```

The length of each *statement set* can be from one to several statements. The **default** portion is optional.

The **switch** works by checking the **variable** against all the constants. As soon as a match is found, that set of statements is executed. If the **break** statement is omitted, then execution continues until the end of the **switch**. Think of **case** as a label. Execution continues until a **break** statement is found, or the **switch** ends.

The following example can be used to process a menu selection:

```
ch = getche();

switch (ch) {
  case 'e': enter();
       break;
  case 'l': list();
       break;
  case 's': sort();
       break;
  case 'q': exit(0);
  default: printf("unknown command\n");
       printf("try again\n");

}
```

typedef

The **typedef** keyword creates a new name for an existing data type. The data type can be either one of the built-in types or a structure or union name. The general form of **typedef** is

```
typedef type_specifier new_name;
```

For example, to use the word **balance** in place of **float**, write

```
typedef float balance;
```

union

The **union** keyword assigns two or more variables to the same memory location. The form of the definition and the way an element is referenced are the same as for **struct**. The general form is

```
union union_name {
   type element1;
   type element2;
   .
   .
   .
   type elementN;
} union variable_name;
```

unsigned

The **unsigned** data-type modifier tells the compiler to eliminate the sign bit of an integer and to use all bits for arithmetic. This doubles the size of the largest integer but restricts it to positive numbers. For example, to declare **big** to be an unsigned integer, write

```
unsigned int big;
```

void

The **void** type specifier is primarily used explicitly to declare functions that return no meaningful value. It is also used to create **void** pointers (pointers to **void**), which are generic pointers capable of pointing to any type of object.

volatile

The **volatile** modifier tells the compiler that a variable may have its contents altered in ways not explicitly defined by the program. These

may include variables that are changed by hardware, such as real-time clocks, interrupts, or other inputs.

while

The **while** loop has the general form

```
while(condition) {
   statement block
}
```

If a single statement is the object of the **while**, the braces can be omitted.

The **while** tests its *condition* at the top of the loop. If the *condition* is FALSE to begin with, the loop will not execute at all. The *condition* can be any expression.

The following example of a **while** loop reads 100 characters from a disk file and stores them in a character array:

```
t = 0;

while(t<100) {
  s[t]=getc(fp);
  t++;
}
```

THE C PREPROCESSOR

C includes several preprocessor commands that give instructions to the compiler. These are examined here.

#define

The **#define** preprocessor command performs macro substitutions of one piece of text for another throughout the file in which it is used. The general form of the directive is

```
#define name string
```

Notice that no semicolon appears in this statement.

For example, if you wish to use TRUE for value 1 and FALSE for value 0, declare the following two macro **#defines**:

```
#define TRUE 1
#define FALSE 0
```

This causes the compiler to substitute 1 or 0 each time TRUE or FALSE is encountered.

#error

The **#error** preprocessor directive forces the compiler to stop compilation when it is encountered. It is used primarily for debugging. Its general form is

```
#error message
```

When **#error** is encountered, C displays the message and the line number.

#include

The **#include** preprocessor directive instructs the compiler to read and compile another source file. The source file to be read in must be enclosed between double quotation marks or angle brackets. For example, the following code instructs the C compiler to read and compile the header for the disk-file library routines:

```
#include "stdio.h"
```

#if, #ifdef, #ifndef, #else, #elif, #endif

These preprocessor directives selectively compile various portions of a program. They are most useful to commercial software houses that provide and maintain many customized versions of one program. The general idea is that if the expression after an **#if**, **#ifdef**, or **#ifndef** is TRUE, the code between one of the preceding directives and an **#endif** is compiled; otherwise it is skipped. The **#endif** directive marks the end

of an **#if** block. The **#else** can be used with any of the above in a manner similar to the **else** in the C **if** statement.

The general form of **#if** is

#if constant expression

If the *constant expression* is TRUE, the block of code is compiled.

The general form of **#ifdef** is

#ifdef name

If the *name* has been defined in a **#define** statement, the block of code following the statement is compiled.

The general form of **#ifndef** is

#ifndef name

If the *name* is currently undefined by a **#define** statement, the block of code is compiled.

For example, here is the way some of the preprocessor directives work together:

```
#define ted 10

main()
{
#ifdef ted
  printf("Hi Ted\n");
#endif
  printf("bye bye\n");
#if 10<9
  printf("Hi George\n");
#endif
}
```

This code prints "Hi Ted" and "bye bye" on the screen, but not "Hi George."

The **#elif** directive creates an **if/else/if** statement. Its general form is

#elif constant-expression

The **#elif** can be used with the **#if**, but not the **#ifdef** or **#ifndef** directives.

THE C STANDARD LIBRARY

Unlike most other languages, C does not have built-in functions to perform disk I/O, console I/O, and a number of other useful procedures. The way these things are accomplished in C is by using a set of predefined library functions supplied with the compiler. This library is usually called the "C Standard Library." Your program can use library functions at your discretion. The compiler automatically links the functions during the link process.

The C language contains a large number of library functions and these are fully described in your C user manual. Also, *C: The Complete Reference,* by Herbert Schildt (Osborne/McGraw-Hill, 1987) discusses the library functions in considerable detail.

TRADEMARKS

AT®	International Business Machines Corporation
Color/Graphics Adapter™	International Business Machines Corporation
CP/M®	Digital Research, Inc.
DEC™ PDP-11™	Digital Equipment Corporation
IBM®	International Business Machines Corporation
IBM Monochrome Adapter™	International Business Machines Corporation
Intel®	Intel Corporation
Microsoft®	Microsoft Corporation
OS/2™	International Business Machines Corporation
PCjr™	International Business Machines Corporation
PS/2®	International Business Machines Corporation
Turbo Pascal®	Borland International, Inc.
UNIX®	AT&T

INDEX

W

Y

The manuscript for this book was prepared and submitted to Osborne/McGraw-Hill in electronic form. The acquisitions editor for this project was Jeffrey Pepper, the technical reviewers were William H. Murray III and Chris H. Pappas, and the project editor was Fran Haselsteiner.

Text design by Pamela Webster, using Palatino for text type and display.

Cover art by Bay Graphics Design Associates. Color separation by Colour Image. Cover supplier, Phoenix Color Corporation. Book was printed and bound by R.R. Donnelley & Sons Company, Crawfordsville, Indiana.

OS/2™ PROGRAMMING:

AN INTRODUCTION

Get up to speed on OS/2™ programming with this fast-paced guide to the operating system developed jointly by Microsoft and IBM.

Schildt, experienced programmer and ace instructor, has distilled months of research into a well-organized and highly readable discussion of OS/2. You'll save countless hours of poring through thousands of pages of OS/2 documentation.

OS/2™ Programming: An Introduction uses many example C programs to explain each topic thoroughly. Schildt emphasizes OS/2 applications as he describes

- Multitasking
- Interprocess communications
- The creation of dynamic link libraries
- Device monitors
- Presentation Manager
- Screen services
- Mouse services
- Keyboard services
- File I/O

OS/2™ Programming: An Introduction begins with an overview of OS/2. It then details the core API services and leads you to more advanced topics. The book also provides an excellent introduction to the Presentation Manager graphical interface. Included are numerous useful examples and sample programs from which you can begin developing your own OS/2 toolkit.

You'll have a solid OS/2 foundation on which to build when you've finished reading **OS/2™ Programming: An Introduction.**

Programming expert Herbert Schildt is the author of numerous best-sellers including: *DOS Made Easy, C: The Complete Reference, C Made Easy, Advanced C, Second Edition, C: Power User's Guide, Artificial Intelligence Using C, Turbo C®: The Complete Reference, Turbo C®: The Pocket Reference, Using Turbo C®,* and *Advanced Turbo C®*. He also also written *Advanced Turbo Prolog®: Version 1.1, Advanced Turbo Pascal®, Modula-2 Made Easy,* and *Advanced Modula-2*. Schildt is president of Universal Computing Laboratories, Inc., a software engineering firm. He holds a master's degree in computer science from the University of Illinois at Champaign-Urbana.

- OS/2 is a trademark of International Business Machines Corp.
- Turbo C, Turbo Pascal, and Turbo Prolog are registered trademarks of Borland International, Inc.

ISBN 0-07-881427-8