Programming MS-DOS for the 1990s

Ray Duncan
Charles Petzold
M. Steven Baker
Andrew Schulman
Stephen R. Davis
Ross P. Nelson
Robert Moote

EXTENDING DOS

EXTENDING DOS

EXTENDING DOS

Edited by Ray Duncan

Ray Duncan
Charles Petzold
M. Steven Baker
Andrew Schulman
Stephen R. Davis
Ross P. Nelson
Robert Moote

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

Production Editor: Amorette Pedersen
Manuscript Editor: Kate Lee Johnson
Technical Consultant: Andrew Schulman
Cover Design by: Copenhaver Cumpston
Illustrators: Eddie O'Brien and Kathy Turok
Set in 10.5-point Palatino by Benchmark Productions

ABCDEFG-MW-943210
First Printing, April 1990

ISBN 0-201-55053-9

Table of Contents

Introduction

As we entered the latter half of the 1980s, there was a widespread sense among PC software developers, analysts, and journalists that DOS's useful life had run its course. Programmers were eagerly looking forward to the availability of protected-mode, multitasking, virtual memory operating systems such as OS/2 and UNIX. *PC Tech Journal,* in an August 1987 article entitled "The Twilight of DOS," went so far as to say "Although it adds some new capabilities, DOS 3.3, unable to provide the multitasking capabilities promised with OS/2, may be this aging operating system's swan song."

In 1990, *PC Tech Journal* is just a memory, but DOS is still very much alive and kicking. In fact, the prophecies of DOS's doom that were popular just a few years back have been totally confounded. DOS accounted for nearly 11 million (61 percent) of the 18 million PC operating systems shipped worldwide in 1989, while Macintosh System 6 shipments were estimated at 1.2 million (6.6 percent), UNIX shipments approximately 440,000 (2.4 percent) and OS/2 merely 125,000 (0.7 percent). The installed base of DOS systems now numbers over 45 million, surpassing all other operating systems combined.

What happened? In part, UNIX and OS/2 simply failed to live up to their advertising. The world of UNIX implementations is still splintered into multiple incompatible variants, and has not yet delivered the much-discussed and much-anticipated source and binary portability that was supposed to arrive with UNIX System V. OS/2 has revealed itself to be much more difficult to program than anyone expected; even Microsoft labored over two years to get its first graphical OS/2 application (Excel) out the door. And both systems demand amounts of RAM, fixed disk storage, and CPU horsepower that exclude their use on most existing PCs.

DOS, on the other hand, has made no promises it can't keep; instead, it has evolved steadily through eight major releases without loss of backward compatibility. DOS is simple enough to be easily understood, small enough to run well on even the most humble 8088 machine with one floppy disk, powerful enough to serve as a respectable networking platform, and versatile enough to handle peripheral devices of every description. In this respect, the entire PC industry owes a tremendous debt to Mark Zbikowski and Aaron Reynolds, the Microsoft programmers who wrote DOS version 2.0 and were responsible for its installable device drivers, hierarchical file system, *stream* file and device I/O, and network hooks.

More germane to this book, DOS's simplicity has also allowed it to be extended in directions never foreseen by its authors. In retrospect, the first wave of "DOS extenders" was the class of application known as Terminate-And-Stay-Resident Utilities (TSRs). TSRs introduced novel new capabilities to the DOS environment, but brought with them novel new types of problems: TSRs battling over hardware interrupts, TSR interactions resulting in mysterious system crashes and, of course, the notorious "RAM-cram," caused by TSRs competing for memory with the very applications they were supposed to enhance.

But TSRs are old news now. This book is about the second wave of DOS extenders: products and programming interfaces that allow software developers to take advantage of the 80286, 80386, and 80486 CPUs without abandoning the DOS installed base and distribution channels. The remedies offered range from increased amounts of fast storage for data (EMS and XMS), to true protected-mode execution with relief of memory constraints for both code and data (286 and 386 DOS Extenders), to DOS-based multitasking environments (DESQview and Windows). Some of these solutions can even be made to work together, by means of the Virtual Control Program Interface (VCPI) and DOS Protected-Mode Interface (DPMI).

We, the authors of *Extending DOS*, recognize that this is a rapidly changing field, and we know that no treatment of such a diverse set of topics can be both exhaustive and timely. We have, however, drawn on our own practical experience to provide you with detailed information on the issues that we consider most important.

We hope that this book will serve you well—both in deciding which strategies for extending DOS to adopt, and which to avoid—and we welcome your comments and suggestions for future editions.

Ray Duncan
Los Angeles, California
March, 1990

About the Contributors

M. Steven Baker is the Editor of *Programmer's Journal*. He has written articles on TSRs, undocumented DOS functions, serial communications, X Window, number crunching, and 80386 development tools. He wrote the chapter on "Safe Memory-Resident Programming" in the Waite Group's *MS-DOS Papers* (Howard Sams, 1988). Baker holds bachelor degrees in Architecture and Electrical Engineering from MIT and Master's degrees in Urban Planning and Architecture from the University of Oregon.

Stephen Randy Davis is the author of two books from M&T Publishing: *DESQview: A Guide to Programming the DESQview Multitasking Environment,* and *Turbo C: The Art of Advanced Program Design, Optimization, and Debugging*. He graduated from Rice University in Houston, Texas. Davis has worked in the defense industry since 1979, and heads a consulting firm, North Texas Digital Consulting, Inc.

Ray Duncan is the author of the Microsoft Press books *Advanced MS-DOS Programming* and *Advanced OS/2 Programming,* and was general editor of *The MS-DOS Encyclopedia,* also from Microsoft Press. He is a contributing editor to *PC Magazine* and *Embedded Systems Programming*. Duncan received a B.A. in Chemistry from the University of California at Riverside and an M.D. from the University of California at Los Angeles. He is the founder and owner of Laboratory Microsystems Inc., a vendor of Forth interpreters and compilers for microprocessors since 1979.

Robert Moote is a co-founder and vice president of software at Phar Lap Software, Inc., and is the author of Phar Lap's 386 | DOS Extender and 386 | VMM products. He received a B.A. in Electrical Engineering and a B.A. in Mathematics from the University of Rochester, and has worked in the microcomputer and minicomputer industry since 1977.

Ross Nelson is the author of *The 80386 Book* from Microsoft Press, as well as several articles for *BYTE* and *Dr. Dobb's Journal.* He has a Computer Science degree from Montana State University, and has been involved with microcomputers for over a dozen years. Currently Manager of Software Engineering at Answer Software, Nelson has worked in the Silicon Valley for the last decade, including two years at Intel Corp.

Charles Petzold is the author of the Microsoft Press books *Programming Windows* and *Programming the OS/2 Presentation Manager*; his latest book, *The OS/2 Graphics Programming Interface,* is scheduled to be published in 1990. He is a contributing editor to *PC Magazine* where he writes about OS/2 in the Environments column, and his work appears frequently in *Microsoft Systems Journal.* Prior to his writing career, Petzold worked ten years for a large insurance company, programming and teaching programming on IBM mainframes and personal computers.

Andrew Schulman is a contributing editor to *Dr. Dobb's Journal,* where he specializes in writing about protected-mode programming. He has also written articles for *BYTE* and *Microsoft Systems Journal.* Schulman is a software engineer responsible for CD-ROM network programming at a large software house in Cambridge, Massachusetts. He has ported several large applications to DOS/16M.

Chapter 1

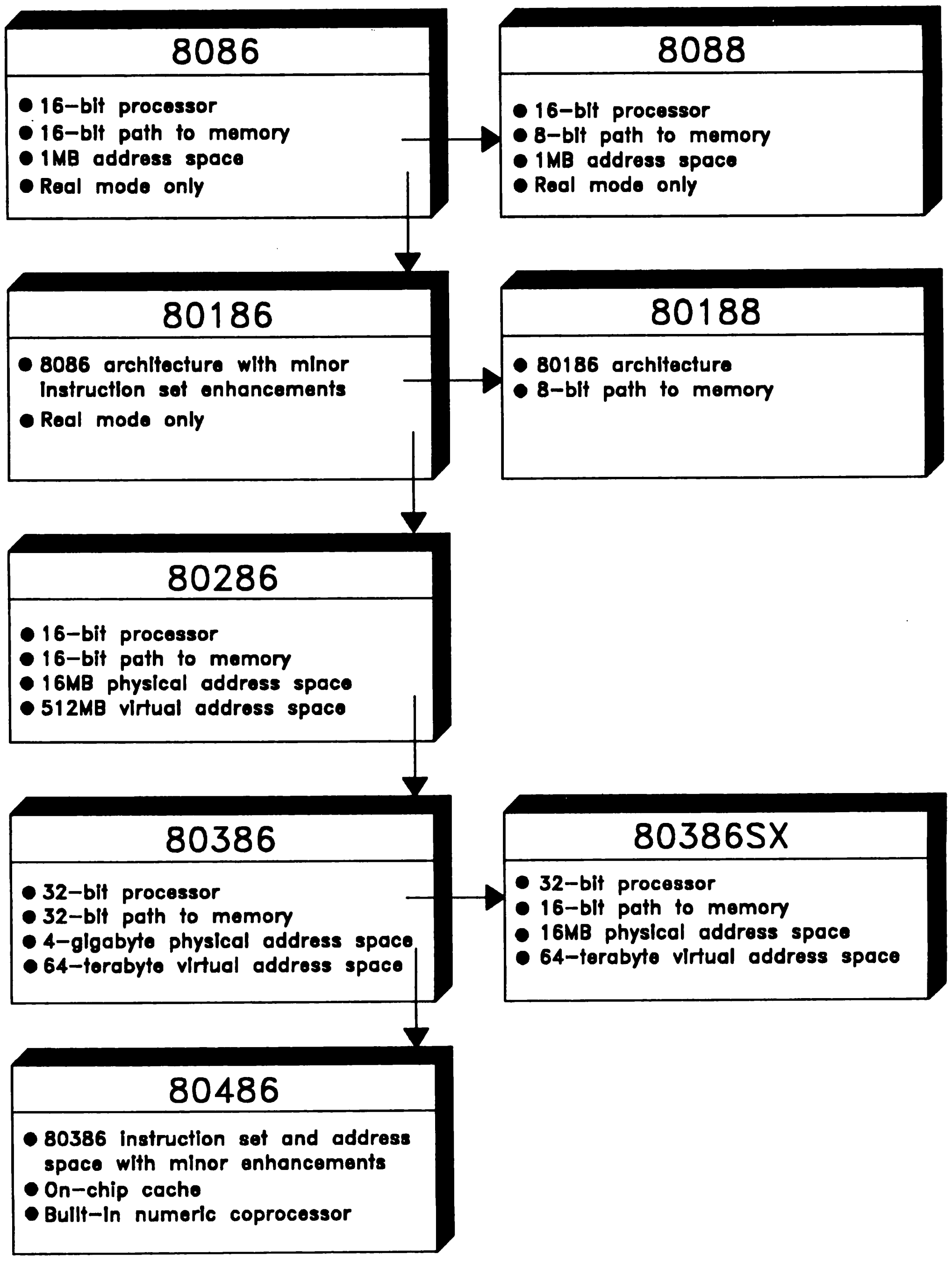
The IBM PC Programming Architecture

Ross Nelson

The advances in microprocessor technology during the last decade have presented software developers with a dilemma. The latest generation microprocessors from Intel—the 80386 and 80486—place mainframe potential in the hands of a single user. But MS-DOS, the operating system on over 90 percent of the personal computers using an Intel microprocessor, was designed three CPU generations ago. Some of the software technology supporting MS-DOS application development is nine years old, and the decisions which shaped that software go back as far as 1974. In addition, the majority of the IBM-compatible PCs in use today—regardless of the CPU they are based on—try to maintain a high degree of hardware compatibility with the original IBM PC based on the Intel 8088.

Taking full advantage of the power of the latest Intel microprocessors would require abandoning DOS and IBM PC compatibility altogether, but this is a luxury few software developers can afford. In order to maintain a presence in the enormous DOS software market, the majority of developers must attempt to satisfy their customers by balancing requests for more performance and more features with the constraints of DOS and the demands of the IBM PC architecture. This chapter will survey the possibilities for extending DOS that are inherent in the architecture of the Intel 80x86 family of processors (see Figure 1-1), in order to set the stage for more detailed treatments throughout the remainder of this book.

Figure 1-1: The Intel 80x86 family tree.



IBM's First Personal Computer

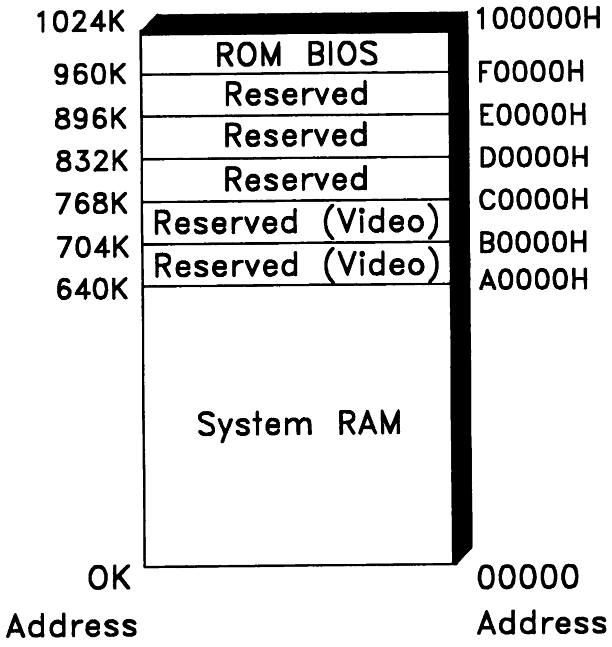
When the IBM PC first appeared in 1981, it was available in two configurations: a "low-end" system with BASIC in ROM, 16K of RAM, and a cassette port for external storage, and a "high-end" system with a 5-1/4-inch floppy disk drive, 64K of RAM, and a disk operating system. Although the first IBM PC may not sound very impressive today, it set several important precedents at the time. It had an *open architecture* with an extensively documented expansion bus; it was built with *off-the-shelf* components and therefore could easily be cloned; and it was based on a 16-bit CPU, leapfrogging the other personal computers of the era which were based on 8-bit microprocessors such as the 8080, Z-80, and 6502.

The central processor processor chosen by IBM for its first PC was the Intel 8088, a slightly slower variant of the 8086. (As all the recent members of the processor family contain the numbers "86," we can reduce confusion by referring to both the 8088 and the 8086 by the designation 8086. The processors are fully software compatible.) The 8086 supported a physical address space of 1024K, or 1 megabyte, but IBM's design restricted the operating system and application programs to the first 640K of the address space, reserving the remaining 384K for use by routines in read-only memory (ROM) and by hardware subsystems. Figure 1-2 shows how the address space was divided.

It cannot be said that the reserved portion of the address space was wasted. The top 64K was used by the ROM BIOS (Basic Input/Output System), a set of routines that provided a standard software interface to essential PC components such as the video display, keyboard, and diskette controller. The ROM BIOS also contained test routines that checked out the PC's hardware when it was turned on or restarted, and a "bootstrap" program that initiated the loading of an operating system from a diskette. The PC's video adapters—the Monochrome Display Adapter (MDA) and Color/Graphics Adapter (CGA)—used the memory addresses `0B0000h` to `0BFFFFh` for RAM *refresh buffers* that controlled the appearance of the display.

As additional subsystems and adapters were introduced, they too were assigned ranges of memory addresses in the reserved 384K area. For example, when the fixed disk controller was introduced in 1982, 16K was allocated for its on-board ROM containing the fixed disk firmware. The Enhanced Graphics Adapter (EGA), which arrived soon afterward, had 16K of on-board ROM too, and also used the memory addresses from `0A0000h` to `0AFFFFh` for its video refresh RAM in high-resolution graphics modes. By the time the PS/2 was announced in 1987, nearly every address in the upper 384K had been spoken for, and the ROM BIOS itself had grown from 64K to 128K.

Figure 1-2: The IBM PC address space.



Although the ROM BIOS supplied a programmatic interface to the hardware, it provided no mechanisms for loading and executing programs, set no standards for disk formats, and had no ability to manage peripheral devices. Those duties fell to the operating system, and in fact the original IBM PC was announced with no less than three different operating systems: Microsoft's MS-DOS, Digital Research's CP/M-86, and Softech's P-System. For various reasons, MS-DOS (marketed by IBM as PC-DOS, and usually referred to as DOS) rapidly became the operating system of choice, and the other two operating systems never achieved any significant base of users.

DOS proved to be another limiting factor in the evolution of personal computers, albeit in ways more subtle than the 640K limit. Ironically, the earliest versions of MS-DOS were patterned closely after Digital Research's CP/M-80 operating system, to aid developers in porting their applications from the 8-bit 8080- and Z-80-based microcomputers that preceded the IBM PC; this resemblance underlies many problems that are still with us today. The first version of DOS, for example, had no programmatic interface for managing memory—when an application was loaded, it could use the entire address space in whatever manner it chose—and the performance of the video display drivers provided by DOS and the ROM BIOS was notoriously poor.

As a result, the halls of software development companies buzzed with PC "folklore" on how to do things faster or better—such as how DOS used certain undocumented locations in memory, the fastest techniques for direct control of the video adapter, and how the serial communications controller could be pushed beyond its documented capabilities. Many of the programs that exploited these non-standard techniques became best-sellers, sometimes because their direct access to the hardware gave them a performance edge unequaled by their competitors. This, in turn, led others to use the same hardware-dependent techniques, all of which would later come back to haunt the manufacturers of PC software and hardware.

The 8086 Becomes a Family

Intel first began shipping its second generation 16-bit microprocessor, the 80286, in 1982. To those who were paying attention, the 80286 represented a significant advance in the capabilities of the microprocessor, and pointed out the path that future generations would take. It extended the physical address space from 1 megabyte to 16 megabytes. It provided for the development of secure multitasking systems, by including a mechanism with which one program could be prevented from corrupting the code or data of another. And it allowed applications to "overcommit" memory, running in a logical address space that was much larger than the physically available memory. It accomplished all this though a mechanism called *protected virtual address mode.*

Before we discuss protected mode in more detail, however, we should quickly review memory addressing on the 8086.

8086 Memory Addressing

On the 8086, a memory address is made up of two parts: a segment and an offset. The 16-bit segment portion of the address, which is loaded into one of the 8086's four segment registers (CS, DS, ES, and SS), is simply multiplied by 16 by the hardware to specify the starting physical address of a block of memory (to make the terminology even more confusing, such a block is also often referred to as a *segment*). The offset, which is likewise a 16-bit value, determines which byte in a block of memory, or segment, is referenced: offset 0 referring to the first byte, offset 1 to the next, and so on. Since the offset can only take on values in the range `0000H` through `FFFFH`, the largest contiguous chunk of memory that can be easily and continuously addressed is 64K—although, since the values in segment registers correspond directly to memory addresses, a program can manipulate these values in order to use larger data structures.

The segment:offset nature of 8086 addressing is actually a remnant of an even earlier architecture. One of the goals of the Intel designers in creating the 8086 was a simple transition from the previous generation, the Intel 8080. On the 8080, all addresses were 16-bit values stored either in a register or as a direct reference in an assembly language instruction. Division of the 8086's 1-megabyte address space into 64K segments allowed a straightforward emulation of the 8080's memory addressing. Programs could be ported directly from the 8080 to the 8086 by setting CS=DS=ES=SS, resulting in a single combined code and data segment and retaining the 16-bit, 64K addressing model. New programs for the 8086 could use 32-bit addresses (both segment and offset) and access an entire megabyte.

The 8086's segmented architecture led to various styles of programming. If a program requires no more than 64K of code and 64K of data, it can load the segment registers once during its initialization, and then ignore them. This style of application is called a *small model* program. The other extreme, called the *large model*, requires the programmer to deal with addresses as 32-bit quantities, loading a segment register with a new value for nearly every memory reference. Most 8086 high-level language compilers support both of these models. Many support other models as well; for example, using 32-bit addresses for code but only 16-bit references for data (*medium model*) or vice versa (*compact model*).

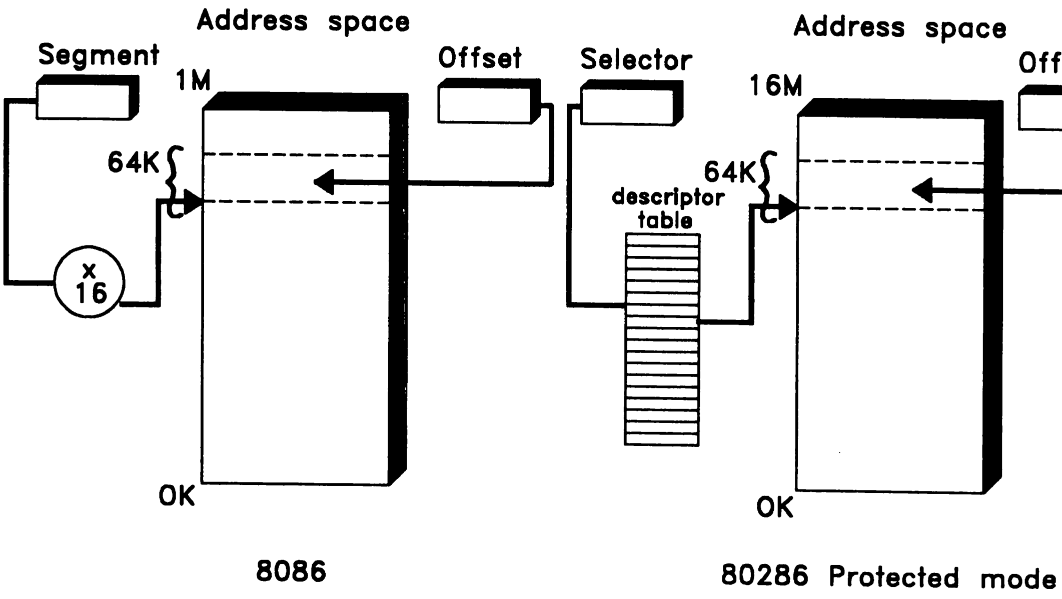
Protected Mode Versus Real Mode

The 80286's protected mode derives its unique capabilities from a change in the way memory addresses are interpreted.

The 80286 CPU starts up in so-called *real mode*, which is basically an 8086 emulation mode; in this mode the 80286 forms addresses in exactly the same manner as an 8086. When the 80286 is switched into protected mode, however, it interprets the contents of a segment register in a radically different way. The value in a segment register is called a selector, and it is used by the CPU hardware as an index into a look-up table—called a *descriptor table*—which contains 24-bit physical base addresses for all the memory segments in the system.

Combination of a 24-bit base address from the look-up table with a 16-bit offset allows the CPU to address 16 megabytes of physical memory. Furthermore, because the same selector and offset (2CA7:0912, for example), may reference any one of many different physical addresses, depending on the base address in the look-up table, the protected-mode selector:offset pair is called a *virtual address*. The addressing methods used by the 8086 and by the 80286 in protected mode are contrasted in Figure 1-3.

Figure 1-3: Addressing modes contrasted.



To recapitulate, a program running on the 8086, or in real mode on the 80286 (or its successors), can read or write any desired memory location at any time, simply by loading an arbitrary value into a segment register. A real-mode operating system cannot monitor or restrict an application program's access to memory, shielding one application from another, because there is no hardware support for such restriction. A protected-mode application, however, can only "see" the memory addresses that the descriptor tables permit it to see. Control over the descriptor tables—and thus the correspondence between values in segment registers and physical memory addresses—is ordinarily reserved to an operating system.

The period immediately following the introduction of the 80286 represents one of the great missed opportunities of the computer industry. If IBM and Microsoft had taken early notice of the 80286's characteristics to the extent of requiring adherence to DOS 2.0's memory management techniques, discouraging programmers from using hard-wired memory addresses and hardware I/O port addresses in their programs, and enhancing DOS and the ROM BIOS with some efficient and flexible video drivers, the transition between the 8086 and the protected-mode operation of the 80286 might have been relatively painless. Instead, direct hardware access techniques became even more entrenched in PC application software, and the design of DOS and the PC's hardware became a captive of the applications' behavior.

Solving Real Problems

Since protected mode didn't really become an issue for most programmers until several years after the PC/AT was introduced, other methods had to be used to squeeze programs into the limited memory supported by DOS. Among these were overlays, expanded memory (LIM EMS memory), and the limited use of extended memory by real-mode programs for storage of data.

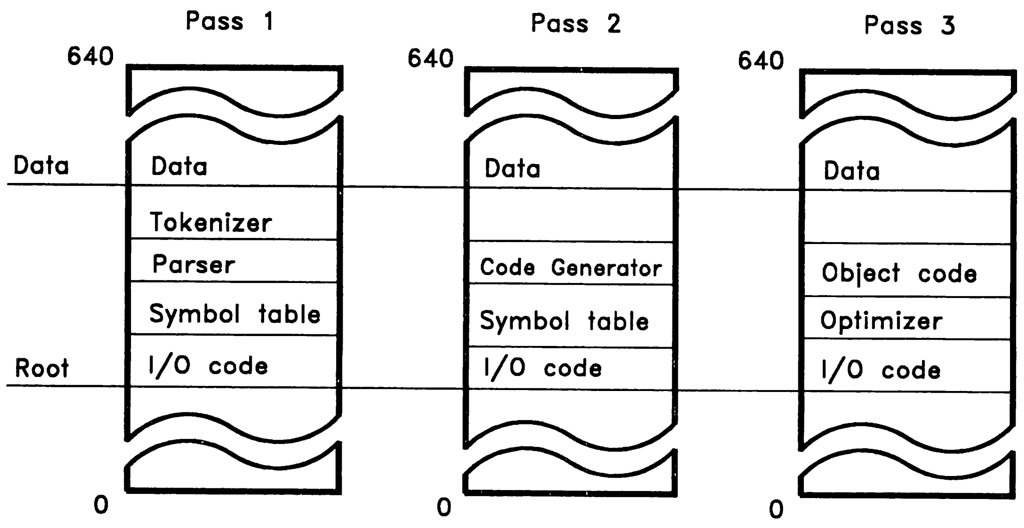
Overlays

The first technique invented to deal with the problem of "too much program and not enough memory" is called *overlaying*. It predates the personal computer by many years, and is best suited to applications that process data in orderly stages, or those in which one of many different possible operations is selected early in the execution process.

An example of a program that might employ overlays is a compiler, which operates, let us say, in three stages or *passes*. The first pass reads in the source program, building the symbol table and checking for syntax errors. It creates a tokenized form of the source for use by the next pass. The second pass operates on the tokenized output of pass one, translating the high-level language to pseudo-assembler output. The third pass performs optimizations and converts the pseudo-assembler code to true object code.

Let us assume that the portion of our hypothetical compiler that performs I/O is used in all three passes, the symbol table functions are used in pass 1 and pass 2, the parser is only used in pass 1, the optimizer is only used in pass 3, and so on. To conserve memory, the parts of the compiler might be organized as shown in Figure 1-4. At any given time, only the code that is necessary for the current phase of the compiler's execution is present in memory; the remainder is stored on disk until it is needed.

Figure 1-4: Overlaid processing.

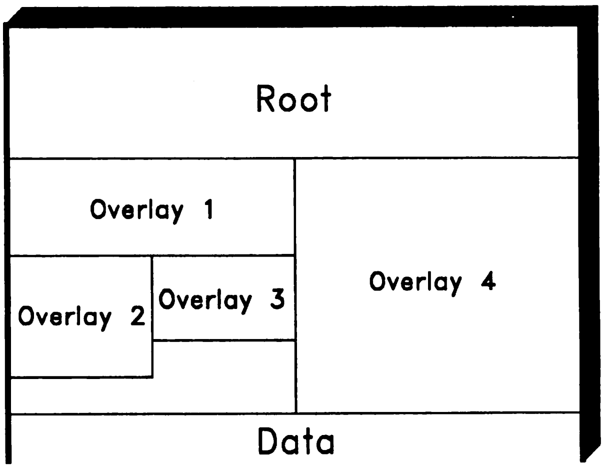


The portion of an overlaid program that is always resident is called the *root*. The portions that replace one another in memory as execution of the program progresses are called *overlays*. In the figure, the I/O code corresponds to the root; the other routines are the overlays. The code fragments that make up an overlay are grouped together and given an overlay name or number.

Overlays are typically built by a linker, which also generates or includes overlay manager code to manage the overlay loading process. The simplest linkers, such as Microsoft LINK, create a root area and a single level of overlays. More sophisticated linkers, such as Phoenix Technology's PLINK86 or Pocket Soft's .RTLINK, can create a hierarchical system of overlays, as shown in Figure 1-5. In this example, overlays 1 and 2, or 1 and 3, can be resident simultaneously. Overlay 4 replaces all other overlays.

Overlaid programs run on any DOS system and generally require no special programming techniques. However, use of overlays has two important drawbacks: the overlay segments must be loaded from secondary storage on disk, which can be quite a slow process; and overlays are useful mainly for programs with a great deal of code and relatively little data, because most overlay managers cannot overlay or swap data. Programs manipulating large amounts of data, like spreadsheets, must find other ways to expand their effective memory space.

Figure 1-5: Hierarchical overlays.



Expanded Memory

Lotus 1-2-3 is the archetypal spreadsheet, and it illustrates the needs of such programs for large amounts of fast storage. The basic concept of a spreadsheet is quite straightforward. The user is presented with a two-dimensional array of locations, or cells, each of which can contain either data or a formula to be evaluated. The more cells in use, the more memory is required.

By 1984, Lotus's customers were building spreadsheet models with thousands of cells, and were running out of memory within the 640K confines of

DOS. Lotus needed a way to add more memory to the 8088. Since the data was accessed frequently, it was necessary to have rapid access, as close as possible to the speed of primary memory. Disk storage was out of the question. Eventually, Lotus worked together with Intel and Microsoft to devise a new species of fast storage: *expanded memory.*

Typically, when you add a memory board to a computer, the RAM addresses are fixed. For example, if you had two boards with 256K RAM each, the first would most likely start at address `00000h`, and the second at address `040000h`. In contrast, the memory on an expanded memory board has no fixed address. Instead, when an expanded memory board is installed, a page frame is chosen—a 64K block within the 384K reserved area that doesn't conflict with other hardware, such as a video adapter or network card. Each 16K chunk, or page, of expanded memory can then be dynamically assigned to an address within the page frame.

Lotus, Intel, and Microsoft also standardized a software interface for expanded memory boards and called it the Expanded Memory Specification, or EMS for short. The interface is typically implemented in a software module called an expanded memory manager, which is provided by the expanded memory board's manufacturer. The manager keeps track of which pages are in use, which may be used by a new application, and which pages are currently accessible. To make use of expanded memory, an application calls the manager to request the number of pages it needs, to make its expanded memory pages available within the page frame as necessary, and finally to release its expanded memory pages before it terminates.

The primary advantage of expanded memory is that it works in any existing PC-compatible computer. An EMS-compatible memory board can be added to either an 8086- or 80286-based system. In newer machines with 80386 or 80486 microprocessors, no special expanded memory hardware is required at all; instead, software emulators use advanced features of these CPUs to implement the EMS standard. The main drawback to the use of expanded memory is that it requires special programming within the application; each page must be explicitly enabled by a call to the expanded memory manager when the data it contains is needed. Further discussion of expanded memory can be found in Chapter 2.

Extended Memory

The first IBM personal computer to incorporate the Intel 80286 CPU was the PC/AT, introduced in 1984. The PC/AT had a true 16-bit bus and the capacity to support the full 16 megabytes of RAM addressable by the 80286. The memory above the 1-megabyte boundary (called *extended memory* by IBM) could only be accessed by a program running in the 80286's protected mode. Realizing that a protected-mode operating system for the PC/AT might be a long time coming, IBM provided real-mode programs with limited access to the extra memory and protected mode in the form of several new ROM BIOS function calls.

The most important of the new ROM BIOS functions, `Int 15h` Function `87h`, places the 80286 into protected mode, copies a block of data from an address anywhere in the 16-megabyte range to any other address, and returns to real mode. This simple function might have contended with EMS as a solution to the data storage problems of spreadsheets and similar programs, but there were a number of obstacles to its success. First, the function was not widely publicized when the PC/AT first appeared; most programmers had to stumble on it while reading the ROM BIOS program listings. The function was also significantly slower than expanded memory; an EMS driver can access a block of memory simply by enabling the required page, but the ROM BIOS function must change the CPU mode twice as well as copying the data back and forth.

The most important weakness of ROM BIOS `Int 15H` Function `87H`, however, is that it assumes a very simple operating model: one program "owns" all of extended memory. The EMS standard, by comparison, allows expanded memory to be shared between applications, TSRs, interrupt routines, and so on. In 1988, a standard called XMS (eXtended Memory Specification) was agreed upon to address ownership and allocation of extended memory blocks by multiple applications, in a manner similar to EMS. The details of programming under the XMS standard are covered in Chapter 3.

Although both EMS and XMS could satisfy a program's needs for large amounts of fast storage, neither proved to be without annoyances. An application has to specifically map or move data in and out of its conventional memory. A program has to deal somehow with data structures that don't fit into the maximum block that can be copied by a single call to `Int 15H` Function `87H`, or into a single expanded memory page (or even the entire expanded memory page frame). Developers of large programs began to sigh longingly, "If we could run in protected mode, we could use all 16 megabytes as regular memory."

Using Protected Mode

The "protection" in protected mode is derived from an "operating system's-eye" view of the world. If you assume that microcomputers are just like mainframes and minicomputers, and as they get faster and more powerful, people will want them to do anything a mainframe or mini can do, you must plan for multitasking.

If the computer is doing many things "simultaneously"—printing one document, editing another, and updating a database, for example—you don't want a bug in one program to affect any of the others. Protected mode isolates one program from another by not allowing direct access to any of the system resources. A level of indirection is imposed on all memory accesses, which can be validated by the operating system. We saw this in Figure 1-3: in protected mode, segment registers contain special values called selectors, which point to a system resource called a *descriptor table*. This table is interpreted by the CPU, but maintained by the operating system.

Under a true protected operating system, application programming is actually simplified. Selectors become just one less thing to worry about. No need to compute addresses or do segment arithmetic; the operating system doles out selectors at load time, or in response to memory allocation requests. Any attempt by an application to use an invalid or inappropriate selector results in a trap (software interrupt) that is serviced by the operating system. The operating system may handle the trap in a variety of ways, the most common being to terminate the offending program.

But the lack of a DOS-compatible operating system to manage the descriptor tables and other system resources tended to put a damper on the development of protected-mode PC software, no matter how desirable it appeared. For those who chose not to wait for a brand new operating system, protected mode created a bit of a mess. Their only option was to use a variation on the method IBM originally provided for memory transfers, that is, to run in protected mode part of the time, and in real mode part of the time. When the application was running and needed access to large amounts of memory, the processor would be in protected mode; when the application needed an operating system service (opening a file, for example), it would switch to real mode so that DOS could handle the request.

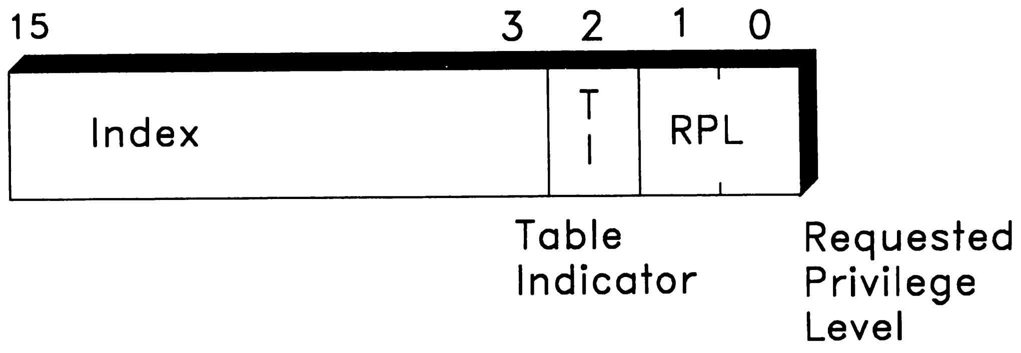
This simple-sounding solution is really quite a technical challenge, because it requires a far deeper understanding of protected mode than an application programmer would typically want or need under a true protected-mode operating system. To get a feel for the steps involved, we must examine protected mode in more detail.

Protected-Mode Details

The 80286 architecture assumes as an underlying model a group of cooperating tasks, supported by a reliable kernel operating system. To prevent intentional or inadvertent damage of one task by another, each task has a separate, *local* address space and access to the system's *global* address space. A privilege mechanism keeps operating system-level code and data secure from outside tampering.

As we have already seen, this entire system was made possible through one key architectural change in the transition between the 8086 and the 80286: the use of indirection in segment addressing. In the 8086, the contents of a segment register are simply multiplied by 16 to generate the base address of a memory segment. In the 80286's protected mode, the selectors found in segment registers are made up of three separate components, as shown in Figure 1-6.

Figure 1-6: A protected-mode selector.



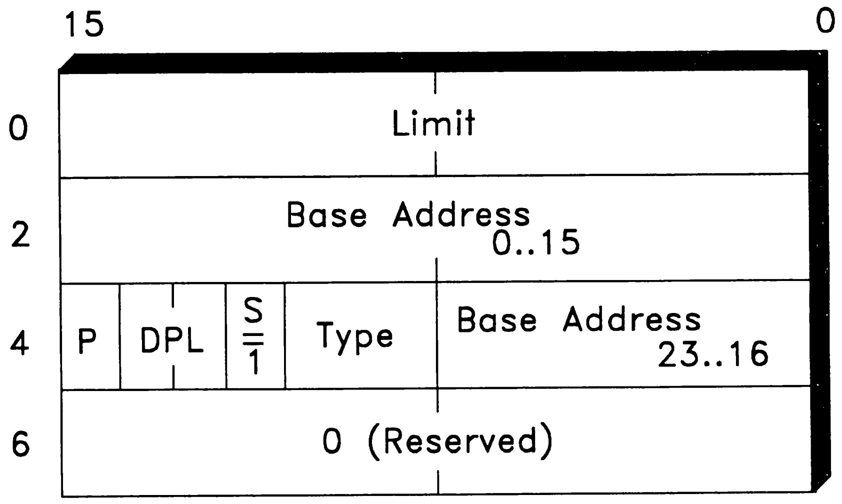
The two low-order bits of the selector make up the Requested Privilege Level, or RPL. The 80286 supports four privilege levels, numbered from zero (most privileged) to three (least privileged). Applications almost universally run at the lowest privilege levels, and all their selectors have an RPL of three. Bit 2 of the selector, the Table Indicator, or TI bit, indicates whether the specified segment comes from the local address space or the global address space. A value of 0 selects global addressing; a 1 selects local addressing. The 13 high-order bits act as an index into a descriptor table. The descriptor table—either a Global Descriptor Table (GDT) or a Local Descriptor Table (LDT), depending on the value of the TI bit—contains information about the segment, including the starting address.

Descriptors are at the heart of protected-mode operation because they fully describe and control all aspects of their corresponding memory blocks, or segments. Each descriptor contains a base (or starting) physical memory address for its segment, the length of the segment, the privilege level required to access the segment, and some bits that define usage attributes of the segment. A descriptor takes up eight bytes. The C data structure used to access individual components of the descriptor is shown below:

```
#pragma  int16
typedef unsigned char byte;
typedef unsigned int word;
typedef unsigned long dword;
struct MEMSEG {
     word     limit;
     word     base_lo;
     struct {
     unsigned base_hi : 8;                // Note:  Ordering of bit fields is
     unsigned type : 4;                   // compiler dependent. Check your
     unsigned s : 1;                      // manual before using this struct.
     unsigned dpl : 2;
     unsigned present : 1;
              } ar;
     word     unused;
     };
end
```

Note that the *base address* found in a descriptor is 24 bits long, comprising the second word of the descriptor, as well as the lower eight bits of the third word; this allows a segment to begin anywhere in the 80286's 16-megabyte address space. The descriptor's *limit* field defines the last legal offset that can be addressed within the corresponding segment. In protected mode, segments are not always 64K in size; the segment size is actually *limit*+1. If a program has only 20K of code, for example, the limit of the code segment descriptor is set accordingly; any attempt to branch beyond the bounds of the segment is automatically detected by the hardware, and causes a special type of interrupt called a *fault*. Figure 1-7 shows the structure of an 80286 segment descriptor in a more diagrammatic fashion.

Figure 1-7: The 80286 segment descriptor.



The *access rights* (AR) byte is located in the upper half of the third word of the descriptor. The fourth word is unused in the 80286 and must contain the value 0. The descriptor for a segment is accessed whenever a selector that points to it is loaded into a segment register, and the access rights byte is the first thing the processor examines. The bits in the AR byte are defined as follows:

- P—Present: This bit must be set to 1 to indicate that the data for this segment is actually present in memory. If P=0, a fault occurs when the selector is loaded.
- DPL—Descriptor Privilege Level: To access a segment, the privilege level of the executing program (called the Current Privilege Level, or CPL) must be equal to or more privileged than the DPL. Attempts to access a descriptor that is more privileged than the executing code result in protection faults.
- S—Segment: This bit is set to 1, indicating a memory segment. When S=0, the descriptor has a slightly different format and is used to define special system constructs.
- TYPE—Type: This field defines additional attributes of the segment. Bit 3 (mask `0x08`) of this field is set to 1 if the segment is an executable segment. Any attempt to write to an executable segment causes a fault. For executable segments, bit 2 (mask `0x04`) is set to 1 to indicate a *conforming* segment

(that is, a segment that changes privilege according to the privilege of the calling routine), and bit 1 (mask 0x02) is set to 1 if the segment may be read as data, as well as executed as code. As you might expect, attempts to read non-readable segments result in protection faults. If bit 3 is 0, the segment is a data segment. In this case, bit 2 is set to 1 to indicate an *expand down* segment (a special segment type for stacks) and bit 1 is set to 1 to mark the segment as *writable*. If bit 1 is 0, the segment is *read-only*; this attribute is enforced via the protection mechanism as well. For both code and data segments, bit 0 (mask 0x01) of the TYPE field is 0 if the descriptor has never been accessed. The hardware sets bit 0 to 1 each time a selector pointing to the descriptor is loaded into a segment register.

Descriptors are grouped into tables, two of which are necessary for the system to operate correctly. The Global Descriptor Table (GDT) contains descriptors that are shared across the entire system, and defines the global address space of the machine when it is in protected mode. The size and starting address of the GDT are defined by values in a special register, GDTR, which must be initialized before entering protected mode. Similarly, the IDTR contains the starting address and size of the Interrupt Descriptor Table (IDT). The IDT helps the system manage interrupts and is analogous to the set of interrupt vectors running from 0000:0000h to 0000:03FFh in real mode. In protected mode, however, the IDT is not restricted to starting at physical address 0, and it contains 8-byte descriptors rather than 4-byte pointers.

As described so far, the protected-mode model is defined by the GDT and IDT, which contain descriptors that define code and data segments. These are not sufficient, however, to support two other 80286 features previously mentioned: multitasking, and a local address space for use by each task. In order to see how these features are implemented, we must look at another class of descriptors, called *system descriptors*. These descriptors are identified by a 0 in the S bit of the AR byte. The two possible formats are shown diagrammatically in Figure 1-8; the code structure is shown below:

```
#pragma  int16
typedef unsigned char byte;
typedef unsigned int word;
typedef unsigned long dword;
struct SYSSEG {
    word     limit;
    word     base_lo;
```

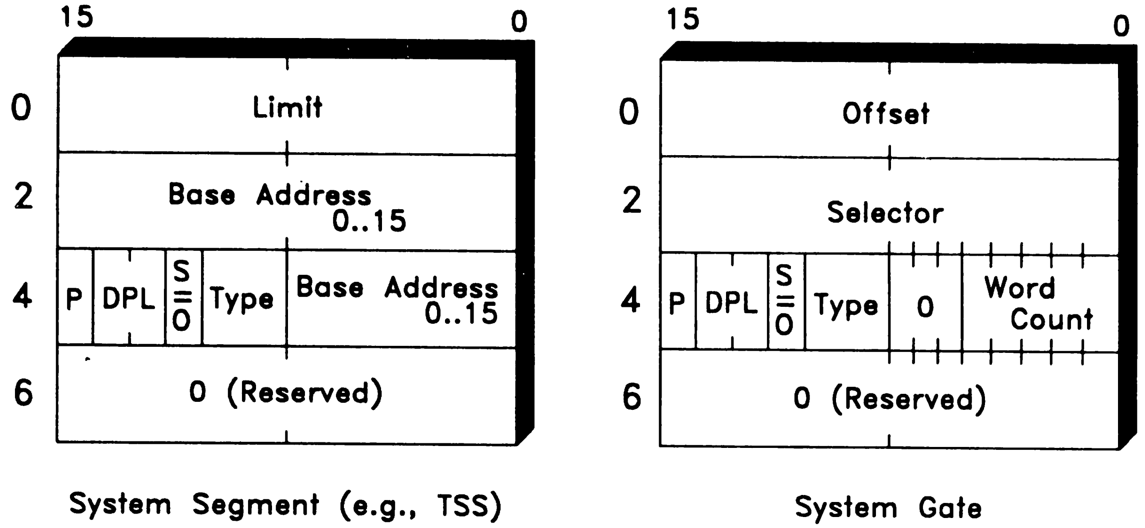
```
     struct {
     unsigned base_hi : 8;              // Note: Ordering of bit fields is
     unsigned type : 4;                 // compiler dependent. Check your
     unsigned s : 1;                    // manual before using this struct.
     unsigned dpl : 2;
     unsigned present : 1;
              } ar;
     word     unused;
     };

struct GATE {
     word     offset;
     word     select;
     struct {
     unsigned wc : 5;
     unsigned unused : 3;
     unsigned type : 4;
     unsigned s : 1;
     unsigned dpl : 2;
     unsigned present : 1;
              } ar;
     word     unused;
     };
```

The Present bit and DPL fields of system descriptors are used in the same manner as in segment descriptors. The TYPE field takes one of the following values, and determines which of the two descriptor formats is being used.

- 0 Invalid descriptor
- 1 Task State Segment (TSS)
- 2 Local Descriptor Table (LDT)
- 3 Busy TSS
- 4 Call gate
- 5 Task gate
- 6 Interrupt gate
- 7 Trap gate.

Descriptor types 1 through 3 have the format described by the SYSSEG structure in the C code above; types 4 through 7 use the GATE structure. A *gate* is a special kind of descriptor that allows transfer of control (via interrupt or subroutine call) between routines executing at different privilege levels. The SYSSEG descriptors look much like memory segments and, in fact, describe areas of memory. Selectors pointing to these descriptors, however, cannot be loaded into segment registers. Editing the contents of a TSS or LDT requires creation of a data segment with the same base address and limit as the system segment. This technique is called *aliasing*.

Figure 1-8: 80286 system descriptors.



The Local Descriptor Table (LDT) and Task State Segment (TSS) are critical to the implementation of multitasking and a local address space. The LDT descriptor points to a descriptor table that is used when the TI bit of a selector is 1, and the LDTR register always contains the selector of the currently active LDT. Each task has its own LDT, so that its private memory is "invisible" to all other tasks, and the operating system changes the value in LDTR as it transfers control of the CPU from one task to another. The TSS contains a copy of all the general and segment registers for a given task. The operating system's dispatcher can switch tasks simply by branching to a TSS. All the registers and flags belonging to the current task are saved in its TSS, and the registers are loaded with the data saved in the new TSS.

A full description of the multitasking capabilities of 80286 protected mode is beyond the scope of this book; it is the addressing capabilities that concern us here. We now have enough information to create a picture of the structures that must be present in memory before protected-mode operation can continue.

Because protected-mode addressing is table-oriented, it is not possible for an application to "manufacture" segment addresses, as it can on the 8086. For example, the address of the video buffer for a color monitor begins at 0B8000h in most

PCs. The real-mode segment value 0B800h points perfectly to the beginning of the buffer. In protected mode, however, the selector value 0B800h is an index into descriptor 1700h of the GDT with a Requested Privilege Level of 0. Only the operating system knows what descriptor is at that index, and any program running at application privilege level (usually 3), and attempting to access a segment at the highest privilege level, will cause a protection fault.

Clearly, programs that run in protected mode must rely on the operating system to give them access to system memory and other hardware resources. "Well-behaved" real-mode applications that already do so will port easily to protected mode. In general, the things that must be avoided include:

- the use of constant or "hard-wired" segment or selector values
- segment or selector arithmetic, or use of segment registers for "scratch" storage
- access to memory not specifically allocated to the application by the operating system
- writing to code segments
- direct port I/O.

Note that access to the I/O ports is restricted in protected mode as well. It's easy to see why, of course. If you assume a multitasking environment, it won't do to have more than one program attempting to control the same device. Requests for device input and output must be routed through the operating system, which can ensure sequential "ownership" of the device.

DOS Extenders

When a DOS-compatible protected-mode operating system failed to arrive in a timely fashion for the 80286, *DOS extenders* appeared instead. DOS extenders are something less than an operating system, but more than a subroutine library. Essentially, they act like an operating system when it comes to memory management features, hiding descriptor table management the way an operating system would, but they have no device handlers or file system. The DOS extender passes an application's requests for these features on to DOS.

The mechanism used to perform this digital legerdemain is called *mode switching*, and it is not something Intel had in mind when the 80286 was first created. The 80286, you may recall, was introduced in 1982, only one year after the IBM PC. Intel assumed that the advantages of protected mode were so apparent that everyone would convert, and real mode would become a distant memory.

Besides, a transition mechanism between the two modes could jeopardize the security of protected-mode operation. Intel didn't realize that almost no one would pay any attention to protected mode until much later, when DOS applications dominated the marketplace.

As a result, the 80286 can only be returned to real mode by resetting the processor. Fortunately, the designers at IBM included a mechanism to perform this reset under software control when they created the PC/AT. They also included an option in the BIOS that provides for the resumption of program execution at a predetermined location after a reset, rather than always booting the operating system and starting from scratch. The combination of these two capabilities allows an 80286 to run in protected mode, reset to real mode, run a specific routine, and reenter protected mode—or vice versa. One of the main functions of a DOS extender is to ensure that these mode transitions are properly managed.

If mode switches happened only at DOS system calls (`Int 21h`), the work of a DOS extender would be relatively straightforward; however, a number of the events in a PC are asynchronous or interrupt-driven. For example, when one of the keyboard's keys is pressed or released, the processor is signalled via an interrupt, and the interrupt handler routine for the keyboard is real-mode code located in the ROM BIOS. Similarly, DOS's date and time are updated by the real-mode interrupt handler for a timer chip interrupt which occurs 18.2 times per second.

The DOS extender must field all interrupts in protected mode, save the processor state, switch into real mode, reissue the interrupt so that it can be serviced by the appropriate interrupt handler, switch back to protected mode, and resume execution of the application. With these details taken care of, however, the application programmer is free to make use of the full protected-mode address space and other features of the 80286. Chapter 4 covers two popular DOS extenders for the 80286: DOS/16M and OS/286.

Intel's 32-bit Microprocessors

The lure of protected mode became even stronger in 1985, when Intel introduced its first 32-bit microprocessor, the 80386. The 80386 was followed in 1987 by the 80386SX, a 32-bit processor with a 16-bit hardware bus, and in 1989 by the 80486, a very fast processor with an on-chip cache and floating-point hardware. From a software point of view, the 80386SX and 80486 are virtually undistinguishable from the 80386. The following description of the 80386, therefore, applies to the 80386SX and 80486 as well.

Like the 80286, the 80386 supports real-mode operation, for the sake of compatibility with DOS and its applications. It also supports all the features of 16-bit protected mode on the 80286. But when the 80386 is running in its preferred, native protected mode, it is a true 32-bit CPU with many new capabilities. All the registers and instructions on the 80386 (with the exception of segment registers and the instructions that manipulate them) can perform their operations 32 bits at a time. 16-bit operations are still supported, so the 32-bit registers have new names to distinguish them in instruction mnemonics. Table 1-1 lists the 16-bit general register names and their 32-bit counterparts.

Table 1-1: 16- and 32-bit general registers.

80286 General Registers	80386 General Registers
AX, BX, CX, DX	EAX, EBX, ECX, EDX
SP, BP, DI, SI, IP	ESP, EBP, EDI, ESI, EIP

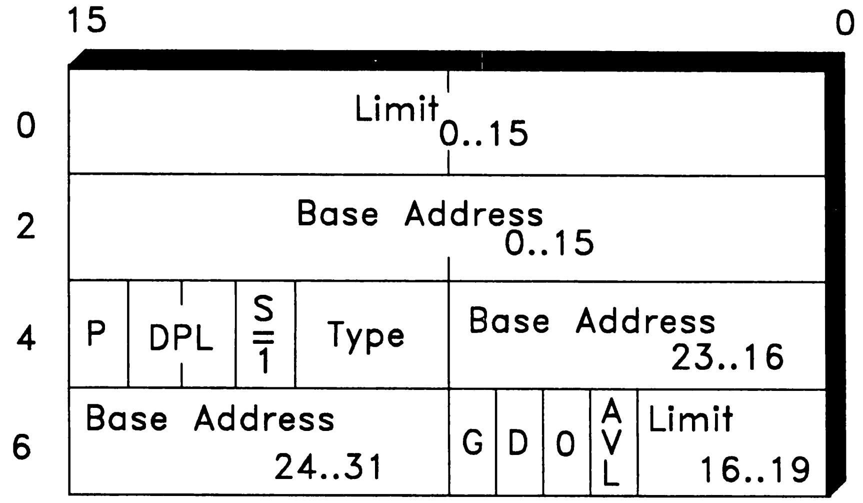
Even more importantly, addressing capabilities are extended on the 80386 too. Though selector values remain 16-bit, using the same GDT- and LDT-based descriptor table look-ups, the offset portion of an address is extended to 32-bits, allowing segment sizes of up to 4,096 megabytes, or 4 gigabytes. The small model of one code segment and one data segment now allows programs to use as much as 8 gigabytes of memory.

Intel achieved these extensions to the programming model without sacrificing 80286 compatibility by making use of the reserved field in the descriptor. Figure 1-9 shows the 80386 descriptor format. (For ease of comparison, the descriptors are shown in 16-bit format. On the 80386, however, only two 32-bit reads are required to load a descriptor, compared with the four 16-bit reads required on the 80286.)

For memory-referencing descriptors, the base address portion has been extended from 24 bits to 32. The limit field is now 20 bits rather than 16, and two bits named "G" and "D" have been added.

The G bit controls the granularity of the limit field. When G=0, the limit field has byte granularity, allowing a maximum segment size of 2^{20}, or 1 megabyte. When G=1, the limit field has 4K or "page" granularity: each increment in the value of the limit increases the maximum segment size by 4,096 bytes. For instance, a page-granular segment with limit=3 contains 16K of data.

Figure 1-9: 80386 segment descriptor.



The D bit determines the default operand and addressing modes. When D=0, segments behave as in 80286 protected mode, that is, instruction operands are 16 bits, and segment offsets may not exceed `0FFFFh`. When D=1, the default operand size is 32 bits, and segment offsets may vary throughout the entire 4 gigabyte range; restricted, of course, by the descriptor's limit value.

The 80386 also introduced a significant change to the familiar 8086 instruction set. In the 8086 and 80286, registers can only be used as base or index registers for memory references in certain combinations, which are listed in Table 1-2. These restrictions apply in both real and protected modes.

Table 1-2: 8086/80286 addressing modes.

Operand	Description
DISP	16-bit displacement (offset)
[BX]+DISP	Base register + displacement
[BP]+DISP	Stack frame + displacement
[SI]+DISP	Source index + displacement
[DI]+DISP	Destination index + displacement
[BX]+[SI]+DISP	Base + index + displacement
[BX]+[DI]+DISP	Base + index + displacement
[BP]+[SI]+DISP	Stack frame + index + displacement
[BP]+[DI]+DISP	Stack frame + index + displacement

As you can see, the registers AX, CX, DX, and SP cannot be used in 8086 or 80286 address computations. In contrast, on the 80386, register addressing is fully generalized, and any of the eight general registers, EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI, may be used. The three fundamental forms of 80386 addressing are shown in Table 1-3.

Table 1-3: 80386 addressing modes.

Operand	Description
DISP	Displacement alone (32-bits)
[REG]+DISP	Base register + displacement
[REG]+[REG*SCALE]+DISP	Base register + scaled index register + displacement

The first two 80386 addressing modes are simply generalized forms of the original 8086 displacement and base-plus-displacement addressing modes. The third 80386 form is like 8086 base-plus-index addressing, except that the index register is automatically multiplied by a scale value of 1, 2, 4, or 8. To illustrate, consider the C language code fragment below and the assembler code that a compiler might generate.

```
/* C language */
int i;
long sum, vector[400];
   ...
sum += vector[i]

; 8086/80286 assembler
I        DW    ?
SUM      DD    ?
VECTOR   DD    400 * (?)
    ...
         MOV    SI, I              ; get index
         SHL    SI, 2              ; scale for long integers
         MOV    AX, VECTOR[SI]     ; fetch array value
         MOV    DX, VECTOR[SI]+2
         ADD    SUM, AX            ; compute sum
         ADC    SUM+2, DX

; 80386 assembler
I        DD    ?
SUM      DD    ?
VECTOR   DD    400 * (?)
    ...
         MOV    EAX, I             ; get index
         MOV    EAX, VECTOR[EAX*4]; fetch indexed array value
         ADD    SUM, EAX           ; compute sum
```

The 8086 or 80286 version uses three registers and includes a separate operation to adjust the index for the operand size. The 80386 version requires only one register and performs the index scaling on the fly.

If you are willing to limit your market to customers who have 80386 machines, there are DOS extenders that allow you to create applications that fully exploit the 32-bit registers, enhanced instruction set, and enormous address space of the processor. Chapters 5 and 8 contain a discussion of these products.

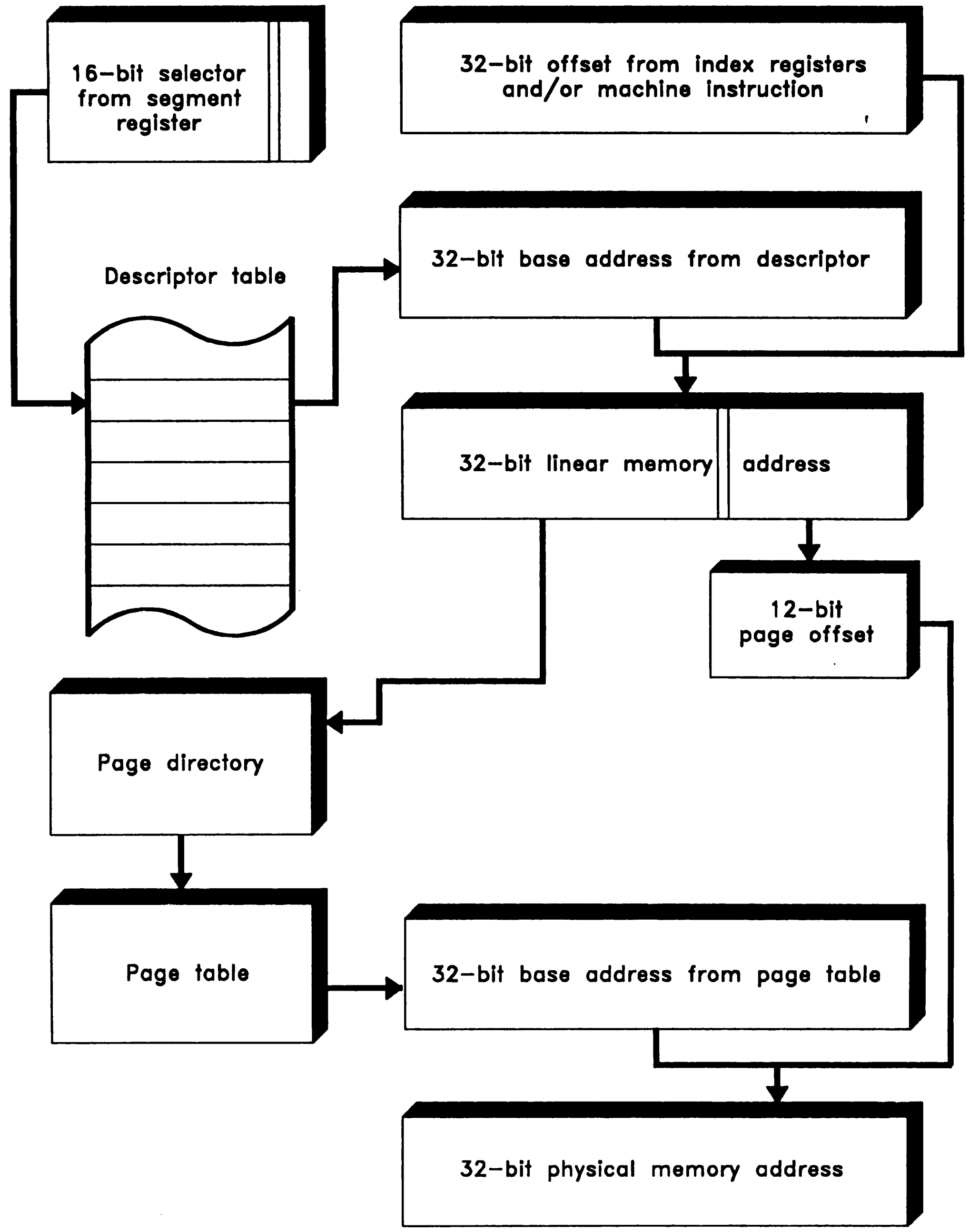
The 80386 has other capabilities too; among them is a new form of address indirection called *paging*. Recall that in protected mode on the 80286, a selector:offset pair is converted to a physical address by fetching the base address from the descriptor table and adding the offset. On the 80386, the base address and offset are combined to form a 32-bit *linear address* which can then be passed through the CPU's paging mechanism to yield the final, physical address. In effect, paging allows each 4K block of RAM to have its own virtual address. Figure 1-10 illustrates a simplified version of what happens on the 80386 when paging is enabled.

The designers of the 80386 added paging to support the needs of high-performance, virtual-memory operating systems. But paging can also be put to use serving DOS applications by emulating EMS hardware with extended memory. Since the 80386's paging operates on 4K boundaries, four 80386 pages can be used to simulate one EMS page. By manipulating the page tables, an EMS emulator can in effect create an EMS page frame—responding to an application's EMS mapping requests by page-table mapping linear addresses within the page frame onto physical addresses above the 1-megabyte boundary.

One 80386 = Many DOS Machines

One of the most interesting features of the 80386 is its *virtual 86 (V86) mode*. As we saw earlier in the chapter, the model underlying protected mode is one of multiple tasks sharing the resources of the processor. The state of each task, that is, the contents of its CPU flags and registers, is stored in the TSS for a suspended task, and in the actual machine registers when the task is running. On the 80386, a bit named "VM" was added to the flags register. When VM=1 in an executing task, the task is executing in virtual 86 mode, and the GDT and LDT are not used. Instead, the selector values are multiplied by 16 (as in real mode) to generate a *linear* address, which is still subject to translation via the paging mechanism. Consequently, it is possible on the 80386 to run more than one real-mode program at a time in separate "V86 machines"; each program under the illusion that it is running in the fixed, 1-megabyte address space of the 8086.

Figure 1-10: Virtual address translation through paging.


16-bit selector from segment register
32-bit offset from index registers and/or machine instruction
Descriptor table
32-bit base address from descriptor
32-bit linear memory address
12-bit page offset
Page directory
Page table
32-bit base address from page table
32-bit physical memory address

Because the paging hardware allows the system to protect memory on a page-by-page basis, an operating system can trap a V86 program's writes to screen memory, possibly redirecting (remapping) the output to a "shadow" buffer. This makes it possible to "window" real-mode applications that do not use the DOS or ROM BIOS video drivers. The 80386 also allows an operating system to selectively intercept a V86 application's I/O port reads and writes. The operating system can then let the I/O operation proceed, divert the I/O to a different port, or even carry out the I/O on behalf of the application. Interrupts that are serviced by the protected-mode operating system can also be "reflected" into a V86 machine for service by the V86 application. This global ability of an 80386 operating system or control program to monitor and control a V86 program's I/O and memory access is known as *device virtualization.*

OS/2 version 2.0, Windows/386, DESQView/386, and some versions of UNIX exploit the 80386's page tables and virtual 86 mode to run multiple DOS applications concurrently. The major weaknesses in this scheme are that each DOS program is confined within its V86 machine to the 640K limit, and that the DOS programs—not being written with multitasking in mind—can't communicate or cooperate effectively with each other.

Operating Environments

We have seen some of the techniques by which a developer can create an application that surpasses DOS-imposed boundaries. Another possibility is to make use of the features provided by operating environments. Operating environments are similar to DOS extenders, but appear more like separate operating systems. They are not complete; however, they reside "on top" of DOS and make use of DOS services and the DOS file system.

Two of the most popular operating environments are DESQview and Windows. These programs are covered extensively in Chapters 6 and 7. They embody two very different philosophies, and place very different requirements on the developer.

DESQview, from Quarterdeck, is the simpler of the two environments. Its primary advantage to the end user is multitasking. Developers get a larger task space for their applications than would be available under DOS. DESQview makes more memory available by managing EMS memory in a very efficient manner. DESQview runs well on any computer with EMS hardware, or on an

80386- or 80486-based system using V86 mode and paging. Very little extra work is required to make an application DESQview-compatible.

Windows, on the other hand, imposes a radically different "world view" on an application. Standard DOS programs are essentially incompatible with Windows; applications must be completely redesigned to use the features Windows provides. In return, Windows offers a great deal to the end user: a uniform graphical interface, multitasking with transparent swapping of applications to expanded memory, "cut and paste" of text and graphics between applications, and more. But these features come at a price. Users will want an 80286 or faster processor in their machine and a high resolution monitor.

Developers pay a price as well; application development time is far longer under Windows than under DOS, and there is a steep learning curve. Applications developed under Windows do, however, have some advantages that DOS programs do not, and device independence is one of the most important. Applications developed for DOS have to deal with display and printer hardware on their own. Software developers must be aware of the popular monitors and printers, and write the appropriate support code for their applications. Windows applications, on the other hand, will work unchanged on any hardware supported by Windows device drivers.

What About OS/2?

The first version of OS/2, IBM, and Microsoft's protected-mode operating system for the 80286, finally arrived in 1987. OS/2's road to acceptance has been long and arduous. It would have been a difficult one even if its only obstacle had been the lack of applications written for it, but there were additional problems as well. A shortage of RAM chips in 1987-1988 made conversion to OS/2 prohibitively costly, the arrival of OS/2's graphical user interface, the Presentation Manager, was delayed until late 1988, and the initial versions of OS/2 could not take advantage of the 80386's 32-bit addressing and paging capabilities.

In 1990, OS/2's role is more clearly defined, and its future appears somewhat brighter. Presentation Manager has been stabilized and is shipped with all versions, a new file system offers much better disk performance than DOS, key applications such as Microsoft Excel and Aldus Pagemaker have become readily available, and support for 80386-specific, 32-bit applications is offered in OS/2 version 2.0. But growth in the installed base of OS/2 continues to be painfully slow. If you choose to develop for OS/2, you should view it as a strategic investment of time and effort that is not likely to pay off for several years.

Choosing Your Market

Though part of DOS's popularity stems from sheer inertia, market dynamics plays a part as well. In 1984, 8086/8088-based machines numbered just under three million and accounted for 99 percent of the PC-compatible market, with 1 percent of the market owned by the 80286-based AT and its clones. Three years later, the 1987 market share of the 8086-class machines had slipped to 69 percent, but the total number of machines was far greater, over 10 million. The 80286 machines had garnered 29 percent of the market with over four and a quarter million units, and the 80386-based computers were just trickling in at 2 percent market share and 300,000 machines sold.

The market for 8086-class PCs is still growing, though at a slower rate. New computers that are 8086-class machines accounted for approximately 15 percent of total sales in 1989. But these new sales are building on a large base of existing machines. None of these machines will ever run a protected-mode application. Even if you choose to develop for the growing population of 80286/386/486 machines, you may still find it advantageous to support the DOS market; although the future may belong to OS/2, DOS is still the operating system of choice today.

For those who plan to continue serving the DOS market, the rest of this book describes a number of specific tools available to help you push beyond the historical limits of DOS. Table 1-4, below, outlines the options available to you, along with their advantages and disadvantages.

Table 1-4: Current options for extending or replacing DOS.

Method	Advantages	Disadvantages
Overlay	Works on any PC.	Does not support "large data" applications well. Not very fast.
EMS	Works on any PC, no special hardware on 386/486 PCs.	Cost of memory board on 286 PCs. Requires special programming.
XMS	No special hardware required. Access to "unused" extended memory.	Requires 80286 or newer CPU. Requires special programming. Not fast.
286 DOS Extender	All protected-mode features available. Transparent access >640K.	Not compatible with OS/2. Requires 286 or newer machine.
386 DOS Extender	32-bit math and addressing. Transparent access >640K.	Not compatible with OS/2. Requires 386 or newer machine.

Method	Advantages	Disadvantages
DESQview	Multitasking and other features without special programming.	Not compatible with OS/2. Requires EMS for best performance.
Windows	Graphical user interface. Device independence. Rapidly expanding user base. Easy port to OS/2 PM.	Slow on 8086-class machines. Steep learning curve.
OS/2	Transparent access >640K. Multitasking, graphical user interface, and networking.	Current user base very small. Steep learning curve. Requires at least 3 megabytes RAM.
UNIX	Well known/liked in academic and workstation markets. Multitasking and networking. Applications portable to other UNIX systems.	Small, specialized user base. Higher hardware and software costs. Little or no binary compatibility.

Chapter 2

Expanded Memory and the EMS

Ray Duncan

Expanded memory is essentially bank-switched memory—fast storage, which can be larger than the CPU's normal address space, and which is subdivided into smaller chunks (called *pages*) that can be independently mapped in or out of the CPU's address space on demand. As a simple approximation, you can think of bank-switched memory as a deck of cards, where different information can be stored on each of the cards, but only the information on the card that is currently at the top of the deck can be read or changed.

Bank-switched memory is not exactly a new concept. It was used extensively on Apple II and S-100 bus computers to overcome the 64K address limitations of their CPUs, and in the earliest days of the IBM PC, bank-switched memory boards called JRAMs were sold in truckloads by a company named Tall Tree Systems. But the particular type of bank-switched memory known as *expanded memory* has been enormously successful because its sponsors defined it as a software interface rather than in hardware.

The origins of the Lotus/Intel/Microsoft (LIM) Expanded Memory Specification (EMS) have already become somewhat apocryphal. The first EMS, developed jointly by Intel and Lotus, was announced and distributed to software developers at the Atlanta Spring Comdex in 1985. For some unknown reason this document was given the version number 3.0. Microsoft, which was looking for a way to relieve Windows' hunger for memory, quickly bought into the EMS concept. After some minor changes, a new specification—version 3.2—was released as a joint effort of Intel, Lotus, and Microsoft in September of the same year.

EMS didn't become an industry standard without a few squeaks of dissent, however. The ink was hardly dry on the EMS before some of the LIM axis' competitors proposed an alternative standard called the AST/Quadram/Ashton-Tate Enhanced Expanded Memory Specification (AQA EEMS). The EEMS was a proper superset of the original EMS, but expanded, with more flexible mapping functions for use in multitasking environments such as DESQview. Fortunately for software and memory board designers everywhere—who already had enough things to worry about—the good sense of the marketplace prevailed, and the AQA EEMS quickly faded into obscurity.

EMS version 3.2 was completely stable for about two years, and by the end of that period it had gained remarkably broad support among both software and hardware manufacturers. Scores of memory expansion boards appeared on the market that could be configured as expanded memory, while the ability to exploit expanded memory turned up in every class of software from spreadsheets to network drivers to pop-up notepads. And, of course, expanded memory became the natural ally of every vendor of a RAMdisk, disk cache, or print spooler. The programmers responsible for maintaining MS-DOS itself, on the other hand, were much slower to take advantage of expanded memory. MS-DOS 4.0, released in 1988, was the first version that recognized expanded memory at all, and it used that memory only for certain private tables and buffers.

In October, 1987, Lotus, Intel, and Microsoft released version 4.0 of the EMS. Version 4.0 supports four times as much expanded memory as version 3.2, as well as many additional function calls for use by multitasking operating systems. In particular, EMS 4.0 theoretically allows an operating system to run multiple applications at the same conventional memory addresses by paging the applications in and out of expanded memory. EMS 4.0 has not yet become generally significant for software developers as of this writing, because full exploitation of its capabilities requires hardware assistance not available on older EMS boards; the vast majority of EMS-aware applications still relies only on the functions available in EMS 3.2. Some of the characteristics of the three versions of EMS are compared in Table 2-1.

Table 2-1: Comparison of the various EMS versions. The number of function calls shown here includes all distinct subfunctions defined in the EMS.

EMS Version	Release Date	Memory Supported	Function Calls	Page Size	Page Mapping
3.0	April '85	8 megabytes	14	16K	above 640K
3.2	September '85	8 megabytes	18	16K	above 640K
4.0	October '87	32 megabytes	58	any size	anywhere

Components of Expanded Memory

It is important not to confuse expanded memory and *extended memory.* Both are frequently available on the same machine; in fact, many memory boards can be set up to provide either expanded memory or extended memory or a mixture of both. But extended memory can only be accessed in the protected mode of the 80286, 386, and 486 processors, whereas expanded memory can be accessed in real mode and therefore can be installed and used on 8086/88-based machines such as the original PC and PC/XT. If you skipped Chapter 1, it may be helpful to go back and review the material in that chapter now, before reading on.

When you install expanded memory in your computer, you are really installing a closely integrated hardware/software subsystem (we'll ignore EMS emulators and simulators for the moment). In most cases, the hardware portion is a plug-in board that has some of the elements of an ordinary memory board and some of an "adapter" for a peripheral device: it has memory chips, to be sure, but it also has I/O ports, which must be written by the CPU to make portions of that memory addressable. On some of the more recent, highly integrated PCs based on the Chips and Technology chip sets, the logic to control expanded memory is located right on the system board and can be configured with the ROM BIOS SETUP utility.

The software component of an expanded memory subsystem is called the Expanded Memory Manager (EMM). It is installed when the system is booted, with a DEVICE= directive in the CONFIG.SYS file, just like a device driver. In fact, an EMM has several of the attributes of a real character device driver: it has a device driver *header,* a routine that can handle a subset of the requests that the DOS kernel likes to make on device drivers, and it has a *logical device name.* This device name is always EMMXXXX0, regardless of who manufactured the expanded memory board or wrote the EMM.

But the device driver aspects of an EMM are really tangential. The EMM's main jobs are to control the expanded memory hardware, to administer expanded memory as a system resource that may be used by many different programs at once, and to service the function calls defined in the EMS. Programs request these expanded memory functions from the EMM directly, via a software interrupt that MS-DOS considers "reserved"; the MS-DOS kernel does not participate in expanded memory management at all.

A summary of the complete EMS interface can be found in Table 2-4 at the end of the chapter. The summary may appear bewildering at first, but for purposes of a typical application program, you can ignore all but the rather small subset of EMS functions that are listed in Table 2-2. This subset is straightforward to use and reasonably symmetric. For example, the EMS function number is always placed in register AH, logical page numbers typically go in register BX, expanded memory handles in register DX, and so on. Control is transferred from the application program to the EMM by executing a software `Int 67H`. All EMS functions indicate success by returning zero in register AH, or failure by returning an error code in AH with the most significant bit set (see Table 2-5 at the end of the chapter).

Table 2-2: Summary of the EMS functions most commonly used in application programs.

Expanded Memory Function	Call With	Returns
Get Status	AH = 40H	AH = status

Get Page Frame Address	AH = 41H	AH = status
		BX = page frame segment

Get Number of Expanded Memory Pages	AH = 42H	AH = status
		BX = available pages
		DX = total pages

Allocate Pages	AH = 43H
	BX = number of pages
		AH = status
		DX = EMM handle

Map Expanded Memory Page	AH = 44H
	AL = physical page
	BX = logical page
	DX = EMM handle
		AH = status

Release Pages	AH = 45H
	DX = EMM handle
		AH = status

Get EMM Version	AH = 46H	AH = status
		AL = version

In short, the general form of an assembly language EMM function call is:

```
mov     ah,function     ; AH = EMS function number
.                       ; load other registers
.                       ; with function-specific
.                       ; values or addresses
int     67h             ; transfer to EMM
or      ah,ah           ; test EMS function status
jnz     error           ; jump, error detected
```

If you prefer to program in C, you can easily request EMS services without resorting to assembly language by means of the `int86()` or `int86x()` functions. The framework for such calls is:

```
#include <dos.h>

union REGS regs;

regs.h.ah = function;         // AH = EMS function number
.                             // load other registers
.                             // with function-specific
.                             // values or addresses
int86(0x67, &regs, &regs);    // transfer to EMM
if(regs.h.ah)                 // test EMS function status
    error();                  // execute if error detected
```

The remainder of the examples in this chapter are provided in assembly language, but you should find it quite straightforward to convert these to the equivalent C code, using the example above as a model.

Obtaining Access to Expanded Memory

When you want to use expanded memory in one of your programs, the first step is to establish whether the EMM is present or not. You can do this by one of two methods: the *open file* method or the *interrupt vector* method.

The "open file" method is so called because it is based on using MS-DOS `Int 21H` Function `3DH` to open the EMM by its logical name—just as though it were a character device or a file. Assuming that the open operation is successful, your program must then make sure that it didn't open a real file with the same name by accident. This unlikely possibility can be ruled out by calling the `Int 21H` Function `44H` (`IOCTL`) subfunctions `0` (get device information) and `7` (get output status). Finally, the program should close the EMM with `Int 21H` Function `3EH` to avoid the needless expenditure of a handle—you can't do anything else with the handle anyway. The procedure for testing for the presence of the Expanded Memory Manager using the DOS open and IOCTL functions is illustrated below:

```
emmname db      'EMMXXXX0',0     ; guaranteed device name for
                                 ; Expanded Memory Manager
        .
        .
        .
                                 ; attempt to "open" EMM...
        mov     dx,seg emmname   ; DS:DX = address of EMM
        mov     ds,dx            ; logical device name
        mov     dx,offset emmname
        mov     ax,3d00h         ; fxn. 3DH = open
        int     21h              ; transfer to MS-DOS
        jc      error            ; jump if open failed

                                 ; open succeeded, make sure
                                 ; it was not a file...
        mov     bx,ax            ; BX = handle from open
        mov     ax,4400h         ; fxn. 44H subfun. 00H =
                                 ; IOCTL get device info.
        int     21h              ; transfer to MS-DOS
        jc      error            ; jump if IOCTL call failed

        and     dx,80h           ; bit 7=1 if char. device
        jz      error            ; jump if it was a file
                                 ; EMM is present, make sure
                                 ; it is available...
                                 ; (BX still contains handle)
        mov     ax,4407h         ; fxn. 44H subf. 07H =
                                 ; IOCTL get output status
        int     21h              ; transfer to MS-DOS
        jc      error            ; jump if IOCTL call failed
        or      al,al            ; test device status
        jz      error            ; if AL=0 EMM not available
                                 ; now close handle ...
                                 ; (BX still contains handle)
        mov     ah,3eh           ; fxn. 3EH = close
        int     21h              ; transfer to MS-DOS
        jc      error            ; jump if close failed
        .
        .
        .
```

The interrupt vector method relies on the fact that an EMM, if it is installed, will necessarily have captured the vector for Int 67H, by placing the address of its EMS function call entry point in the vector. An application program testing for the presence of an EMM can simply fetch the contents of the vector, then determine whether the segment portion of the vector points to a device driver header that contains the logical device name EMMXXXX0. Example code for testing for the presence of the Expanded Memory Manager by inspection of the EMM's interrupt vector and device driver header can be found below:

```
emmint  equ     67h             ; Expanded Memory Manager
                                ; software interrupt

emmname db      'EMMXXXX0'      ; guaranteed device name for
                                ; Expanded Memory Manager

        .
        .
        .
        xor     bx,bx           ; first fetch segment from
        mov     es,bx           ; EMM interrupt vector
        mov     es,es:[(emmint*4)+2]

                                ; assume ES:0000 points
                                ; to base of the EMM...
        mov     di,10           ; ES:DI = address of name
                                ; field in driver header
        mov     si,seg emmname  ; DS:SI = EMM driver name
        mov     ds,si
        mov     si,offset emmname
        mov     cx,8            ; length of name field
        cld
        repz    cmpsb           ; compare names...
        jnz     error           ; jump if driver absent
        .
        .
        .
```

Which method you choose for detecting the presence of the EMM depends on the type of program you are writing. For conventional application programs, the open file method is preferred, because it is "well-behaved"—it relies only on standard MS-DOS function calls, and is thus least likely to run into conflicts with TSRs, device drivers, interrupt handlers, or multitasking program managers. The interrupt vector method is considered "ill-behaved" because it involves inspection of memory not owned by the program. But when you are designing a device driver that uses expanded memory, you must employ the interrupt vector method, for reasons that will be explained later in the chapter.

Once your program has established that an EMM is present, it should call the EMM's "get status" function (`Int 67H` Function `40H`) to make sure the expanded memory hardware is present and functional. After all, the fact that the EMM itself is installed doesn't guarantee that the associated hardware is also installed (although most EMMs do abort their own installation if the hardware is missing). It is also appropriate for your program to call the "get EMM version" function (`Int 67H` Function `46H`) at this point, to make sure that all of the EMS functions that it intends to use are actually supported by the resident EMM.

Next, your program should call the "get number of pages" function (`Int 67H` Function `42H`) to determine how much expanded memory is available. This function returns both the total number of physically installed pages, and the number of pages that have not already been allocated to other programs. In most cases the two numbers will be the same, unless your program is running under a multitasking program manager alongside other applications that use expanded memory, or unless TSRs or device drivers that use expanded memory have been previously loaded.

If the number of free expanded memory pages is less than your program needs or expects, it must decide whether to continue execution in a "degraded" fashion or simply display an advisory message and terminate. If there are sufficient pages available, however, the program can proceed to call the "allocate EMS pages" function (`Int 67H` Function `43H`) for the necessary amount of expanded memory. The EMM's allocation function returns an "EMS handle"—a 16-bit value that symbolizes the program's expanded memory pages, and must be used in all subsequent references to those pages. This handle is exactly analogous to the file or device handles you are already accustomed to from your previous MS-DOS programming experience.

The last step in obtaining expanded memory resources is to call the EMS "get page frame address" function (`Int 67H` Function `41H`). This function returns the segment, or paragraph, address of the base of the EMM's page frame—the area used by the EMM to map logical expanded memory pages into conventional memory. The page frame address never changes after the system is booted, so you only need to request it once, during your program's initialization code.

A typical sequence of testing EMM status, allocating some EMS pages, and fetching the page frame address is demonstrated below:

```
pneeded equ     4                   ; number of EMS pages needed

pframe  dw      0                   ; page frame address
tpages  dw      0                   ; total EMS pages
```

```
apages  dw      0               ; available EMS pages
handle  dw      0               ; handle for allocated pages
        .
        .
        .
        mov     ah,40h          ; get EMS system status
        int     67h             ; transfer to EMM
        or      ah,ah           ; check for EMM error
        jnz     error           ; jump, error occurred

        mov     ah,46h          ; check EMM version
        int     67h             ; transfer to EMM
        or      ah,ah           ; check for EMM error
        jnz     error           ; jump, error occurred
        cmp     al,32h          ; make sure EMS 3.2+
        jb      error           ; jump if EMS 3.0

        mov     ah,42h          ; get number of EMS pages
        int     67h             ; transfer to EMM
        or      ah,ah           ; check for EMM error
        jnz     error           ; jump, error occurred
        mov     tpages,dx       ; save total EMS pages
        mov     apages,bx       ; save available EMS pages
        cmp     bx,pneeded      ; enough pages available?
        jb      error           ; jump, too few pages

        mov     ah,43h          ; allocate EMS pages
        mov     bx,pneeded      ; number of pages needed
        int     67h             ; transfer to EMM
        or      ah,ah           ; check for EMM error
        jnz     error           ; jump, pages not allocated
        mov     handle,dx       ; got pages, save handle

        mov     ah,41h          ; get page frame address
        int     67h             ; transfer to EMM
        or      ah,ah           ; check for EMM error
        jnz     error           ; jump, error occurred
        mov     pframe,bx       ; save segment of page frame
        .
        .
        .
```

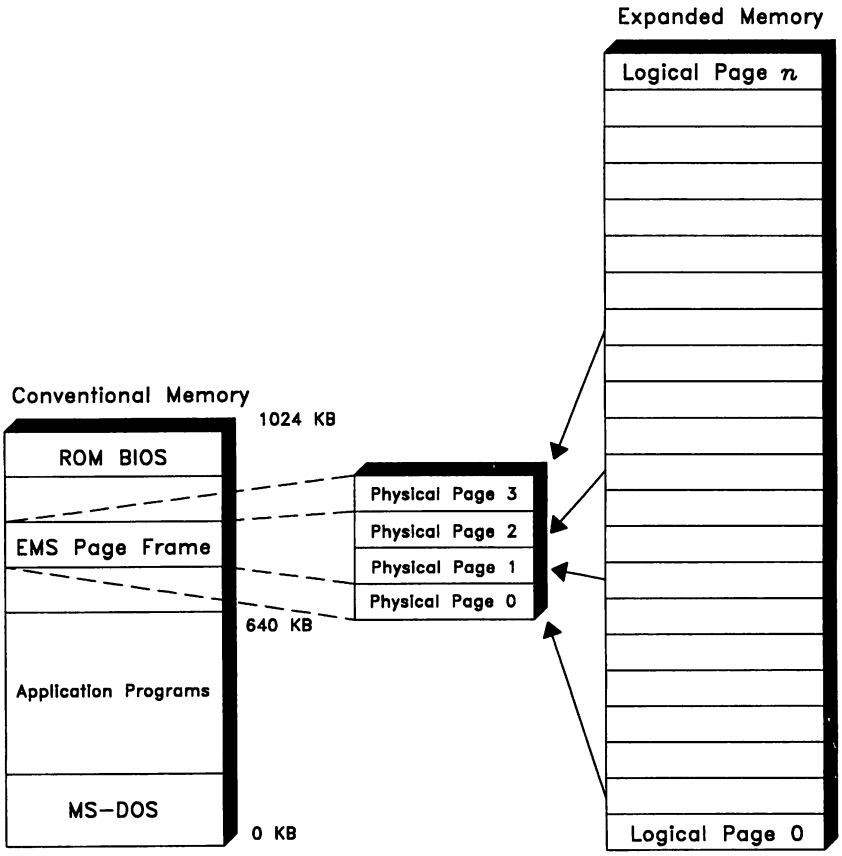
Using Expanded Memory

The logical EMS pages owned by a program and associated with a particular EMS handle are numbered *0* through *n-1*, where *n* is the total number of pages originally allocated to that handle. In EMS 3.0 and 3.2, the pages are always 16K in length. EMS 4.0 allows pages of other sizes to be allocated and used, but it is best to avoid use of "nonstandard" page sizes so that your program will be compatible with the broadest possible range of EMS hardware and software.

A program gains access to the contents of one of its expanded memory pages by calling the "map EMS page" function (`Int 67H` Function `44H`). The EMM accomplishes the mapping by writing commands to the I/O ports on the expanded memory board; logic on the board then ties the memory chips that hold the data for that logical page to the address and data lines of the system bus, so that the logical page becomes visible in the CPU's normal address space.

In EMS 3.0 and 3.2, a mapped page is always made available in the EMM's page frame, which is in turn located at unused addresses above the 640K boundary. The page frame is 64K long, and is divided into four 16K physical pages numbered 0 through 3; thus, a maximum of four different logical pages can be simultaneously accessible. This mapping process is diagrammed in Figure 2-1.

Figure 2-1: Diagram of the relationship between expanded memory and conventional memory.



In EMS 4.0, an EMS 3.x-compatible page frame is supported for compatibility reasons, but the page frame may be larger than 64K, and there may also be multiple page frames. EMS 4.0 also allows pages to be mapped below the 640K boundary on demand, if the proper hardware support is present. Again, it is best to avoid use of these capabilities—which are peculiar to EMS 4.0—unless your program absolutely cannot be made to work without them.

Once a logical page has been mapped to a physical page, it can be inspected and modified with the usual CPU memory instructions. When dealing with the standard EMS 3.x page frame and page sizes, address calculations are straightforward. A far pointer to a mapped page, which is composed of a segment and offset, is built up as follows. The page frame base address returned by `Int 67H` Function `41H` is already in a form that can be loaded directly into a segment register. The offset portion of the far pointer is obtained by multiplying the physical page number (`0-3`) by 16,384 (`4000H`), and adding a logical page displacement in the range 0-16,383 (`0-3FFFH`).

For example, if the address returned by the "get page frame address" function is `D000H`, and logical page 1 for a particular EMM handle has been mapped to physical page 3, then the data in that logical page can be accessed at physical memory addresses `D000:C000H` through `D000H:FFFFH`. The process of mapping a logical page to a physical page, calculating the memory address of the page, and writing data to it, is demonstrated below:

```
pagelen equ     4000h           ; standard EMS page size

pframe  dw      0               ; page frame address
logpage dw      1               ; logical page number
phypage dw      3               ; physical page number
handle  dw      0               ; handle for EMS pages

        .
        .
        .
        mov     ah,44h          ; map EMS page
        mov     bx,logpage      ; logical page 1
                                ; physical page 3
        mov     al, byte ptr phypage
        mov     dx,handle       ; EMS handle
        int     67h             ; transfer to EMM
        or      ah,ah           ; check for EMM error
        jnz     error           ; jump, error occurred
```

```
                                ; form far pointer to page
        mov     es,pframe       ; ES = page frame segment
        mov     ax,pagelen      ; calculate offset of
        mul     phypage         ; physical page in frame
        mov     di,ax           ; let ES:DI = page address

        xor     ax,ax           ; now zero out the
        mov     cx,pagelen      ; mapped page
        rep stosb
```

This code fragment assumes the page frame address was fetched with a call to `Int 67H` Function `41H` earlier in the program's execution, and that a valid EMS handle was previously obtained with a call to `Int 67H` Function `43H`.

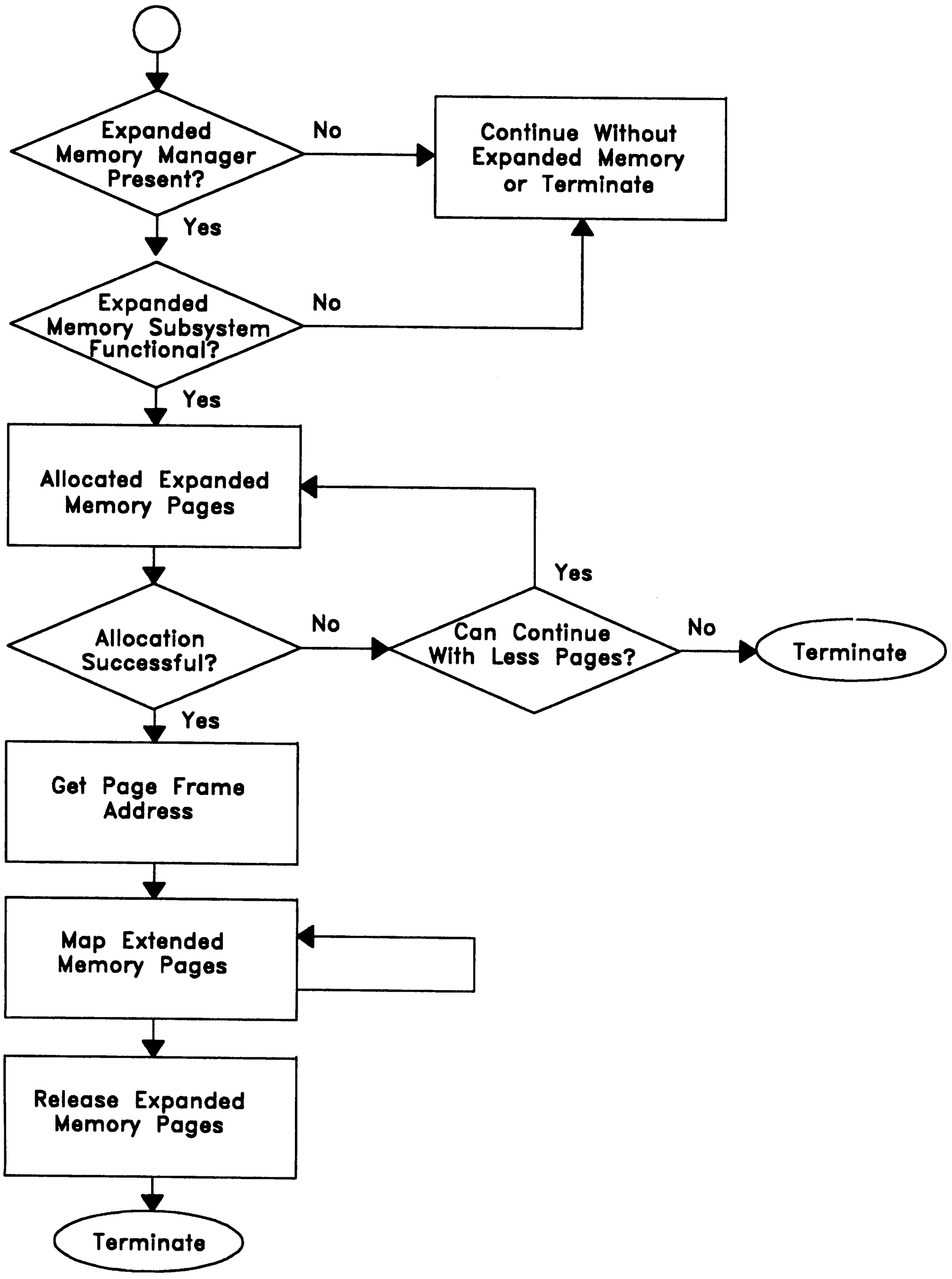
In programs that take advantage of EMS 4.0's ability to support more than four physical pages or more than one page frame, it may be preferable to use a lookup technique to translate physical page numbers to far pointers. Your program can call EMS `Int 67H` Function `58H` Subfunction `00H` to get a list of the physical page numbers and their physical memory addresses.

It is often helpful to think of the expanded memory owned by a program as a sort of virtual file with a length of *n**16,384 bytes (where *n* is the number of allocated EMS pages). To access a particular piece of data in the file, the program first performs a "seek" operation to the nearest sector boundary: it divides the byte offset of the data within the virtual file by 16,384 to find the logical page number, and maps that logical page to a physical page. The remainder, when the byte offset is divided by 16,384, is the offset of the data within the logical page, which can be combined with the page frame base address and the offset of the physical page within the page frame to form a far pointer in the manner described.

When your program is finished using expanded memory, it must be sure to deallocate its EMS handle and logical pages by calling the "release EMS handle" function (`Int 67H` Function `45H`) before terminating. If it fails to do so, the expanded memory owned by the program will be lost and unavailable for use by other programs until the system is restarted. MS-DOS cannot clean up a program's expanded memory resources automatically at termination because MS-DOS does not participate in the expanded memory management in the first place. For the same reason, programs using EMS should contain their own critical error and Control-C handlers, so they cannot be terminated unexpectedly.

A sketch of the entire process of using expanded memory in an application is shown in Figure 2-2.

Figure 2-2: General procedure for expanded memory usage by an application program.


Expanded Memory Manager Present?
No
Continue Without Expanded Memory or Terminate
Yes
Expanded Memory Subsystem Functional?
No
Yes
Allocated Expanded Memory Pages
Yes
Allocation Successful?
No
Can Continue With Less Pages?
No
Terminate
Yes
Get Page Frame Address
Map Extended Memory Pages
Release Expanded Memory Pages
Terminate

EMS Pitfalls for Drivers and TSRs

Using expanded memory in a device driver or TSR utility is somewhat more problematic than in a normal application program. You must concern yourself not only with the logic of your own program, but with protecting yourself against every possible use (and misuse) of expanded memory by other active programs, whether they be drivers, TSRs, or applications.

When a driver or TSR gets control, the state of the system is, by nature, sensitive and unpredictable. In the first place, the driver or TSR was undoubtedly called as the direct or indirect result of the user's interaction with an application program, and that application may well be using expanded memory too. It is crucial that any use of expanded memory by the foreground application not be disturbed. Therefore, it is vitally important that the driver or TSR save the expanded memory mapping context (the association of specific logical expanded memory pages with physical locations in the CPU's address space) at entry, and restore that same context before it exits.

In each successive revision of the EMS, new functions have been defined for saving and restoring the expanded memory subsystem state (summarized in Table 2-3). In EMS version 3.0, Functions 47H and 48H provided all-or-nothing capability on a per-handle basis. If the driver or TSR owned only one expanded memory handle, then it could save only one mapping context at a time. In EMS version 3.2, Function 4EH (which actually consists of four distinct subfunctions) was added, allowing a program to selectively save and restore as many mapping contexts as it had memory to put them in. This made things a lot easier for multitasking program managers, since it allowed them to associate a mapping context with each active application.

Table 2-3: Summary of the EMS functions related to saving and restoring the expanded memory mapping context.

Expanded Memory Function	Call With	Returns	EMS Version

Save Page Map	AH = 47H
	DX = EMM handle
		AH = status	3.0

Restore Page Map	AH = 48H
	DX = EMM handle
		AH = status	3.0

Save Page Map	AH = 4EH
	AL = 00H
	ES:DI = buffer
		AH = status	3.2

Expanded Memory Function	Call With	Returns	EMS Version

Restore Page Map	AH = 4EH
	AL = 01H
	DS:SI = buffer
		AH = status	3.2

Save and Restore Page Map	AH = 4EH
	AL = 02H
	DS:SI = restore buffer
	ES:DI = save buffer
		AH = status	3.2

Get Size of Page Map Information	AH = 4EH
	AL = 03H
		AH = status
		AL = size (bytes)
			3.2

Save Partial Page Map	AH = 4FH
	AL = 00H
	DS:SI = map list
	ES:DI = buffer
		AH = status	4.0

Restore Partial Page Map	AH = 4FH
	AL = 01H
	DS:SI = buffer
		AH = status	4.0

Get Size of Partial Page Map Information	AH = 4FH
	AL = 02H
	BX = number of pages
		AH = status
		AL = size (bytes)
			4.0

In EMS version 4.0, the number of expanded memory pages that can be simultaneously mapped into conventional memory is much larger, and the overhead of saving and restoring the complete mapping state has grown proportionately. Consequently, Function `4FH` was added to manipulate partial mapping contexts. Version 4.0 also defines a host of other new functions directly or indirectly related to page mapping, ranging from mapping of multiple pages with one call (optionally followed by a jump or call to code within the pages) to support for multiple sets of hardware mapping registers. These functions are intended primarily for use by operating systems, so we won't discuss them further here.

When you are writing a device driver or TSR, you must also concern yourself with a difficult issue that doesn't arise in normal MS-DOS application programming: the lack of availability of MS-DOS services after your program is originally installed. A device driver is allowed to use a limited number of MS-DOS `Int 21H` functions *during* installation, but none at all thereafter. As for TSRs, they are typically activated during a hardware interrupt (such as the reception of a keystroke), and since the state of MS-DOS at the time of the interrupt cannot be known in advance, they must rely on undocumented flags and structures to determine whether MS-DOS function calls can be made safely. This all implies that your driver or TSR should perform all the status checks it can, and acquire all the expanded memory resources it expects to ever need, at the time it gets loaded—because interaction with the user at any later point (even to display an error message) will be much more difficult.

One last potential problem we should mention is that the amount of stack space available at the time your TSR or driver is invoked is indeterminate. MS-DOS itself uses three different stacks, depending on the type of function call in progress; applications customarily have their own stacks, whose depth is totally at the discretion of the developer; interrupt handlers often switch to their own stacks; and last but not least, the amount of stack space required by EMS functions may vary from one Expanded Memory Manager (EMM) to another as well as from version to version. The safest strategy is for your driver or TSR to always switch to its own generously sized stack, before using any EMS functions.

EMS Emulators

From the very beginning, the Expanded Memory Specification was formulated strictly as a software interface, without hardware dependence of any kind (other than the assumption that the software is running on an Intel 80x86 CPU). We'll never know whether this was just a happy accident or a stroke of genius on the part of the original Lotus/Intel/Microsoft designers, but the payoff is the same in any event: the nature of the Expanded Memory Specification allows expanded memory functionality to be provided on systems that do not contain any expanded memory hardware at all. Programs that provide expanded memory services in the absence of expanded memory hardware are called *expanded memory emulators* or *simulators*, and they fall into three classes: disk-based EMS emulators, 286 extended memory-based EMS emulators, and 386/486-specific memory managers that export the EMS interface.

Disk-based EMS emulators, such as Turbo EMS or Above Disk, support the EMS `Int 67H` interface, but store the data in allocated EMS pages in a swap file on disk. When an application requests an EMS page to be mapped into the page frame, the emulator reads the EMS page's data in from the swap file and makes it available in RAM. Disk-based EMS emulators will run on any type of PC, from the original 8088-based model on up, but have two severe disadvantages: they are very slow compared to true EMS based on bank-switched memory, and the page frame is almost always located low in conventional memory rather than above the 640K boundary. The unusual location of the page frame causes trouble for some application programs that are not completely well-behaved in their use of EMS services.

80286 extended memory-based EMS emulators are similar to disk-based EMS emulators, in that they typically create a page frame in conventional memory below the 640K boundary. However, these emulators are drastically faster than disk-based emulators because they store the data in the simulated EMS pages in extended memory rather than on disk, using ROM BIOS `Int 15H` Function `87H` to move the data between extended memory and the page frame when a mapping is requested by an application program. On older PC/ATs and clones, such an extended memory-based EMS emulator may allow you to gain the benefits of EMS without purchasing any additional hardware. However, better performance will be obtained by reconfiguring the system's extended memory as expanded memory when the hardware allows it, or by purchasing and plugging in a new board that can supply true expanded memory.

A 386/486-based memory manager such as Qualitas's 386-to-the-Max, Quarterdeck's QEMM, and Microsoft's Windows/386, implements EMS emulation by taking on the role of a little operating system. The memory manager itself runs in the 386/486's 32-bit protected mode, and MS-DOS and its application programs run under the memory manager's supervision in Virtual 86 Mode. This arrangement gives the memory manager complete control over the address space seen by MS-DOS and other real-mode programs; it can use the 386/486 page tables to make any 4K physical memory page appear at any address within the Virtual 86 Machine. For example, 386-to-the-Max can remap extended memory into the "holes" between video adapters and the ROM BIOS, so that device drivers and TSRs can be loaded above the 640K boundary.

Use of Virtual 86 Mode also allows a 386/486-based memory manager to intercept software interrupts, which puts it into a position to simulate EMS memory without the real-mode application's knowledge or cooperation. The memory manager uses extended memory for storage of EMS pages, and simply uses the 386/486 page tables to map the simulated EMS pages on demand into a simulated EMS page frame within the Virtual 86 Machine. The speed of EMS emulation by 386/486-based memory managers is uniformly excellent, because page table manipulation and mode switching on 80386/486 CPUs is very fast. Some memory managers provide additional capabilities as well: 386-to-the-Max actually exports all three of the important software interfaces for memory management: EMS, XMS (Chapter 3), and the VCPI (Chapter 8), while Windows/386 can set up multiple Virtual 86 Machines and perform true preemptive multitasking of MS-DOS applications.

Programming Example: The EMSDISK.SYS Driver

In order to provide a practical example of expanded memory usage by an application program, TSR, or device driver, I've included the source code for a simple EMS-aware RAMdisk (virtual disk) called EMSDISK.ASM. EMSDISK demonstrates the procedure for testing for the existence and functionality of expanded memory that must be used by a driver or TSR. It contains examples of expanded memory allocation, mapping, and the saving and restoring of mapping contexts.

For maximum portability, EMSDISK does not attempt to take advantage of the features of EMS version 4.0; it only relies on functions that are available in EMS version 3.2. Furthermore, EMSDISK.SYS is a simple program as device drivers go; it contains only the essential routines (initialization, build BPB, media check, read, and write) that allow MS-DOS to recognize it as a valid block device.

You may find it helpful to consult a general text on MS-DOS (such as my own book *Advanced MS-DOS Programming, 2nd Edition*) for further information about device driver structure and components.

To assemble and link the file EMSDISK.ASM into the executable device driver EMSDISK.SYS, enter the following commands:

```
MASM EMSDISK;
LINK EMSDISK;
EXE2BIN EMSDISK.EXE EMSDISK.SYS
DEL EMSDISK.EXE
```

(The Linker will display the message *Warning: No Stack Segment*. This warning can be ignored.) To install EMSDISK, add the line:

```
DEVICE=EMSDISK.SYS  nnnK
```

to your CONFIG.SYS file and reboot the system. Make sure that the DEVICE= line for EMSDISK.SYS follows the DEVICE= line that loads your expanded memory manager (such as EMM.SYS for Intel Above Boards). The logical drive identifier that will be assigned to EMSDISK depends on the number of block devices that are already present in the system at the time EMSDISK is loaded.

The parameter `nnnK` on the DEVICE= directive is the desired size of the RAMdisk in kilobytes. If this parameter is missing, or is larger than the amount of free expanded memory, EMSDISK will use all of the expanded memory that is available. For example, if fixed disk drive C: is currently the last drive in your system, you could create a 1-megabyte virtual disk drive D: by adding the following line to CONFIG.SYS:

```
DEVICE=EMSDISK.SYS 1024K
```

When EMSDISK is loaded, it will display a sign-on message and, under DOS 3.0 or later, its drive identifier. If EMSDISK can't find a previously loaded expanded memory manager, or is unable to allocate or initialize its expanded memory pages, it will abort its own installation with an error message.

EMS Example Program

```
; EMSDISK.ASM --- Expanded Memory RAMdisk
; Copyright (C) 1989 Ray Duncan
;
; To build:     MASM EMSDISK;
;               LINK EMSDISK;
;               EXE2BIN EMSDISK.EXE EMSDISK.SYS
;               DEL EMSDISK.EXE
;
; To install:   copy EMSDISK.SYS to the root directory of your
;               boot disk, then add the line
;
;                       DEVICE=EMSDISK.SYS nnnK
;
;               to the CONFIG.SYS file.  This line must follow
;               the DEVICE= line that loads the Expanded Memory
;               Manager.  The parameter nnn is the desired
;               RAMdisk size in KB.  If nnn is missing or zero,
;               all available expanded memory is used.
;
_TEXT   segment public 'CODE'

        assume  cs:_TEXT,ds:_TEXT,es:_TEXT

        org     0

maxcmd  equ     24                      ; maximum driver command code

cr      equ     0dh                     ; ASCII carriage return
lf      equ     0ah                     ; ASCII line feed
blank   equ     020h                    ; ASCII space code
tab     equ     09h                     ; ASCII tab character
eom     equ     '$'                     ; end of message signal

emm_int equ     67h                     ; EMM software interrupt
```

```
psize   equ     16384                   ; bytes per EMS page
ssize   equ     512                     ; bytes per sector
dsize   equ     256                     ; entries in root directory

spp     equ     psize/ssize             ; sectors per page

request struc                           ; request packet template
rlength db      ?                       ; length of request packet
unit    db      ?                       ; unit number
command db      ?                       ; driver command code
status  dw      ?                       ; driver status word
reserve db      8 dup (?)               ; reserved area
media   db      ?                       ; media descriptor byte
address dd      ?                       ; memory address for transfer
count   dw      ?                       ; byte/sector count
sector  dw      ?                       ; starting sector number
request ends                            ; end request packet template

                                        ; device driver header
header  dd      -1                      ; link to next driver in chain
        dw      0                       ; driver attribute word
        dw      strat                   ; "Strategy" entry point
        dw      intr                    ; "Interrupt" entry point
        db      1                       ; number of units, this device
        db      7 dup (0)               ; reserved area

rqptr   dd      ?                       ; address of request packet

savesp  dw      0                       ; save MS-DOS kernel's SS:SP
savess  dw      0

availp  dw      0                       ; logical EMS pages available
totalp  dw      0                       ; total EMS pages in system
ownedp  dw      0                       ; RAMdisk size in EMS pages
pframe  dw      0                       ; segment address of page frame
handle  dw      0                       ; expanded memory handle
dosver  db      0                       ; MS-DOS major version no.

xfrsec  dw      0                       ; current sector for transfer
xfrcnt  dw      0                       ; sectors already transferred
xfrreq  dw      0                       ; number of sectors requested
xfraddr dd      0                       ; working address for transfer

array   dw      bpb                     ; BPB pointer array

bootrec equ     $                       ; EMSDISK boot record
        jmp     $                       ; phony JMP instruction
        nop
        db      'IBM  3.3'              ; OEM identity field
                                        ; BIOS Parameter Block (BPB)
bpb     dw      ssize                   ; 0   bytes per sector
        db      0                       ; 2   sectors per cluster
        dw      1                       ; 3   reserved sectors
        db      1                       ; 5   number of FATs
        dw      dsize                   ; 6   root directory entries
        dw      0                       ; 8   total sectors
        db      0f8h                    ; 0AH media descriptor
        dw      0                       ; 0BH sectors per FAT
br_len  equ     $-bootrec               ; length of boot record

        even                            ; force word alignment
        dw      128 dup (0)
stk     equ     $                       ; local stack for device driver

;
; Driver 'strategy' routine; called by MS-DOS kernel with
; ES:BX pointing to driver request packet.
;
strat   proc    far

        mov     word ptr cs:rqptr,bx    ; save address of request packet
        mov     word ptr cs:rqptr+2,es
        ret                             ; back to MS-DOS kernel

strat   endp

;
; Driver 'interrupt' routine, called by MS-DOS kernel immediately
; after call to 'strategy' routine to process I/O request.
;
intr    proc    far

        push    ax                      ; save general registers
        push    bx
        push    cx
        push    dx
        push    ds
        push    es
        push    di
        push    si
        push    bp
        mov     ax,cs                   ; make local data addressable
        mov     ds,ax
        mov     savess,ss               ; save DOS's stack pointers
        mov     savesp,sp
        mov     ss,ax                   ; set SS:SP to point to
        mov     sp,offset stk           ; driver's local stack

        les     di,rqptr                ; let ES:DI = request packet
        mov     bl,es:[di.command]      ; get BX = command code
        xor     bh,bh
        cmp     bx,maxcmd               ; make sure it's legal
        jle     intr1                   ; jump, function code is ok
        mov     ax,8003h                ; set Error bit and code
        jmp     intr3                   ; for "unknown command"

intr1:  or      bx,bx                   ; is it init call? (function 0)
        jz      intr2                   ; yes, skip save of context
        mov     ah,47h                  ; fxn 47h = save page mapping
        mov     dx,handle               ; EMM handle for this driver
        int     emm_int                 ; transfer to EMM
        or      ah,ah                   ; jump if EMM error while
        jnz     intr5                   ; saving page mapping context

intr2:  shl     bx,1                    ; form index to dispatch table
        call    word ptr [bx+dispch]    ; branch to command code routine
                                        ; should return AX = status
        les     di,rqptr                ; restore ES:DI = request packet

intr3:  or      ax,0100h                ; merge Done bit into status,
        mov     es:[di.status],ax       ; store into request packet
        mov     bl,es:[di.command]      ; was this initialization call?
        or      bl,bl
        jz      intr4                   ; yes, skip restore of context

        mov     ah,48h                  ; fxn 48h = restore page mapping
        mov     dx,handle               ; EMM handle for this driver
        int     emm_int                 ; transfer to EMM
        or      ah,ah                   ; jump if EMM error while
        jnz     intr5                   ; restoring page mapping

intr4:  mov     ss,savess               ; central exit point
        mov     sp,savesp               ; restore DOS kernel's stack
        pop     bp                      ; restore general registers
        pop     si
        pop     di
        pop     es
        pop     ds
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret                             ; back to MS-DOS kernel
```

```
intr5:                                  ; catastrophic errors come here
        les     di,rqptr                ; ES:DI = addr of request packet
        mov     es:[di.status],810ch    ; set Error bit, Done bit, and
        jmp     intr4                   ; error code for general failure

intr    endp

;
; Dispatch table for device driver command codes
;
dispch  dw      init                    ;  0 = initialize driver
        dw      medchk                  ;  1 = media check on block device
        dw      bldbpb                  ;  2 = build BIOS parameter block
        dw      error                   ;  3 = I/O control read
        dw      read                    ;  4 = read from device
        dw      error                   ;  5 = non-destructive read
        dw      error                   ;  6 = return current input status
        dw      error                   ;  7 = flush device input buffers
        dw      write                   ;  8 = write to device
        dw      write                   ;  9 = write with verify
        dw      error                   ; 10 = return current output status
        dw      error                   ; 11 = flush output buffers
        dw      error                   ; 12 = I/O control write
        dw      error                   ; 13 = device open        (DOS 3.0+)
        dw      error                   ; 14 = device close       (DOS 3.0+)
        dw      error                   ; 15 = removeable media   (DOS 3.0+)
        dw      error                   ; 16 = output until busy  (DOS 3.0+)
        dw      error                   ; 17 = not used
        dw      error                   ; 18 = not used
        dw      error                   ; 19 = generic IOCTL      (DOS 3.2+)
        dw      error                   ; 20 = not used
        dw      error                   ; 21 = not used
        dw      error                   ; 22 = not used
        dw      error                   ; 23 = get logical device (DOS 3.2+)
        dw      error                   ; 24 = set logical device (DOS 3.2+)


;
; Media Check routine (command code 1).  Returns code indicating
; whether medium has been changed since last access.
;
medchk  proc    near

        mov     byte ptr es:[di+14],1   ; set "media not changed" code
        xor     ax,ax                   ; return success status
        ret
```

```
medchk  endp

;
; Build BPB routine (command code 2).  Returns pointer to valid
; BIOS Parameter Block for logical drive.
;
bldbpb  proc    near

        mov     word ptr es:[di+20],cs  ; put BPB address in packet
        mov     word ptr es:[di+18],offset bpb
        xor     ax,ax                   ; return success status
        ret

bldbpb  endp

;
; Read routine (command code 4).  Transfers logical sector(s)
; from RAMdisk storage to specified address.
;
read    proc    near

        call    setup                   ; set up transfer variables

read1:  mov     ax,xfrcnt               ; done with all sectors yet?
        cmp     ax,xfrreq
        je      read2                   ; jump if transfer completed
        mov     ax,xfrsec               ; get next sector number
        call    mapsec                  ; and map it
        jc      read4                   ; jump if mapping error
        les     di,xfraddr              ; ES:DI = requestor's buffer
        mov     si,ax                   ; DS:SI = RAMdisk address
        mov     ds,pframe
        mov     cx,ssize/2              ; transfer logical sector from
        cld                             ; RAMdisk to requestor
        rep movsw
        push    cs                      ; restore local addressing
        pop     ds
        inc     xfrsec                  ; advance sector number
        add     word ptr xfraddr,ssize  ; advance transfer address
        inc     xfrcnt                  ; count sectors transferred
        jmp     read1                   ; go do another sector

read2:                                  ; all sectors successfully
        xor     ax,ax                   ; transferred, return ok status

read3:  les     di,rqptr                ; get address of request packet
        mov     bx,xfrcnt               ; poke in actual transfer count
        mov     es:[di.count],bx        ; (in case an error aborted
        ret                                 ; the transfer early)

read4:  mov     ax,800bh                    ; come here if mapping error,
        jmp     read3                       ; return read fault error code

read    endp

;
; Write (command code 8) and Write with Verify (command code 9)
; routine.  Transfers logical sector(s) from specified address
; to RAMdisk storage.
;
write   proc    near

        call    setup                       ; set up transfer variables

write1: mov     ax,xfrcnt                   ; done with all sectors yet?
        cmp     ax,xfrreq
        je      write2                      ; jump if transfer completed
        mov     ax,xfrsec                   ; get next sector number
        call    mapsec                      ; and map it
        jc      write4                      ; jump if mapping error
        mov     di,ax                       ; ES:DI = RAMdisk address
        mov     es,pframe
        lds     si,xfraddr                  ; DS:SI = requestor's buffer
        mov     cx,ssize/2                  ; transfer logical sector from
        cld                                 ; requestor to RAMdisk
        rep movsw
        push    cs                          ; restore local addressing
        pop     ds
        inc     xfrsec                      ; advance sector number
        add     word ptr xfraddr,ssize      ; advance transfer address
        inc     xfrcnt                      ; count sectors transferred
        jmp     write1                      ; go do another sector

write2:                                     ; all sectors successfully
        xor     ax,ax                       ; transferred, return ok status

write3: les     di,rqptr                    ; get address of request packet,
        mov     bx,xfrcnt                   ; poke in actual transfer count
        mov     es:[di.count],bx            ; (in case an error aborted
        ret                                 ; the transfer early)

write4: mov     ax,800ah                    ; mapping error detected,
        jmp     write3                      ; return write fault error code

write   endp
```

```
;
; Dummy routine for command codes not supported by this driver.
;
error   proc    near

        mov     ax,8103h                ; return error code 3
        ret                             ; indicating 'unknown command'

error   endp

;
; Map into memory a logical "disk" sector from the EMS
; pages allocated to the RAMdisk.
;
; Call with:    AX    = logical sector number
;
; Returns:      CY    = clear if no error
;               AX    = offset within EMS page frame
;               AX,CX,DX destroyed
;
;               CY    = set if EMM mapping error
;               AX,CX,DX destroyed
;
mapsec  proc    near

       mov     dx,0                    ; divide sector no. by sectors
       mov     cx,spp                  ; per page, to get EMS page number
       div     cx                      ; now AX=EMS page,DX=rel. sector
       push    dx                      ; save sector within page
       mov     bx,ax                   ; BX <- EMS page number
       mov     ax,4400h                ; fxn 4400h = map phys. page 0
       mov     dx,handle               ; EMM handle for this driver
       int     emm_int                 ; transfer to EMM
       or      ah,ah                   ; test for EMM error
       jnz     maps1                   ; jump, EMM error detected
       pop     ax                      ; get relative sector in page
       mov     cx,ssize                ; relative sector * size =
       mul     cx                      ; offset into EMS logical page
       clc                             ; return CY=clear for no error
       ret                             ; back to caller

maps1: add     sp,2                    ; EMM mapping error detected
       stc                             ; clear stack and return CY=set
       ret                             ; to indicate error

mapsec  endp
```

```
;
; Set up to perform Read or Write subfunction by copying
; requestor's buffer address, starting sector, and sector
; count out of request packet into local variables.
;
setup   proc    near

        push    es                      ; save request packet address
        push    di
        mov     ax,es:[di.sector]       ; initialize starting sector
        mov     xfrsec,ax
        mov     ax,es:[di.count]        ; initialize sectors requested
        mov     xfrreq,ax
        les     di,es:[di.address]      ; initialize requestor's
        mov     word ptr xfraddr,di     ; buffer address
        mov     word ptr xfraddr+2,es
        mov     xfrcnt,0                ; initialize transfer count
        pop     di                      ; restore request packet address
        pop     es
        ret

setup   endp

;
; Initialization routine, called at driver load time.  Returns
; address of 'init' label to MS-DOS as start of free memory, so
; that memory occupied by 'init' and its subroutines is reclaimed.
;
init    proc    near

init1:  mov     ax,3000h                ; fxn 30h = get DOS version
        int     21h                     ; transfer to MS-DOS
        mov     dosver,al               ; save major version number

init2:  xor     ax,ax                   ; check if EMM driver present
        mov     es,ax                   ; if EMM is present, address in
        mov     bx,emm_int*4            ; vector points to EMM driver.
        mov     es,es:[bx+2]            ; now ES:0000 = EMM header
        mov     di,10                   ; ES:DI = addr of device name
        mov     si,offset emm_name      ; DS:SI = name to match
        mov     cx,8                    ; length of device name
        cld
        repz cmpsb                      ; compare EMM name
        jz      init3                   ; jump if name matched
        mov     dx,offset msg1          ; if name didn't match,
        jmp     abort                   ; driver is absent, exit

init3:  mov     ah,40h                  ; fxn 40h = get EMM status
```

```
        int     emm_int                 ; transfer to EMM
        or      ah,ah                   ; check for EMM error
        jz      init4                   ; jump, driver is OK
        mov     dx,offset msg2          ; EMM is non-functional,
        jmp     abort                   ; error message and exit

init4:  mov     ah,46h                  ; fxn 46h = get EMM version
        int     emm_int                 ; transfer to EMM
        or      ah,ah                   ; check for EMM error
        jz      init5                   ; jump, no error
init45: mov     dx,offset msg3          ; error occurred, display
        jmp     abort                   ; error message and exit

init5:  cmp     al,030h                 ; must be version 3.0+
        jae     init6                   ; jump if version is OK
        mov     dx,offset msg6
        jmp     abort                   ; wrong EMM version, exit

init6:  mov     ah,41h                  ; fxn 41h = get page frame
        int     emm_int                 ; transfer to EMM
        or      ah,ah                   ; check for EMM error
        jnz     init45                  ; error occurred, exit
        mov     pframe,bx               ; save page frame segment

        mov     ah,42h                  ; fxn 42h = get no. of pages
        int     emm_int                 ; transfer to EMM
        or      ah,ah                   ; check for EMM error
        jnz     init45                  ; error occurred, exit
        mov     totalp,dx               ; save total EMS pages
        mov     availp,bx               ; save available EMS pages
        mov     ownedp,bx               ; default allocated=available
        or      bx,bx                   ; any pages available?
        jnz     init7                   ; yes, proceed
        mov     dx,offset msg4          ; no pages left, exit
        jmp     abort

init7:                                  ; get KB from DEVICE= line
        les     di,rqptr                ; ES:DI = request packet
        lds     si,es:[di+18]           ; DS:SI = CONFIG.SYS text

init71: lodsb                           ; scan for end of driver name
        cmp     al,blank
        ja      init71                  ; loop while within name
        dec     si                      ; point to delimiter and
        call    atoi                    ; convert KB size parameter
        push    cs                      ; make our data addressable
        pop     ds
        or      ax,ax                   ; size parameter missing?
        jz      init74                  ; yes, use available pages
        mov     dx,ax                   ; save copy of KB
        mov     cx,4                    ; divide KB by 16 to get
        shr     ax,cl                   ; requested EMS pages
        and     dx,0fh                  ; round up needed?
        jz      init73                  ; jump if multiple of 16 KB
        inc     ax                      ; round up to next page

init73: cmp     ax,availp               ; requested > available?
        ja      init74                  ; yes, use available
        mov     ownedp,ax               ; no, save requested pages

init74: mov     ah,43h                  ; fxn 43h = allocate pages
        mov     bx,ownedp               ; no. of pages to request
        int     emm_int                 ; transfer to EMM
        or      ah,ah                   ; check for EMM error
        jz      init8                   ; jump if allocation OK
        mov     dx,offset msg5          ; allocation failed, display
        jmp     abort                   ; error message and exit

init8:  mov     handle,dx               ; save EMM handle for pages
        call    makebpb                 ; set up BIOS Parameter Block
        call    format                  ; format the RAMdisk
        jnc     init9                   ; jump if format was OK
        mov     dx,offset msg7          ; error during formatting,
        jmp     abort                   ; display error and exit

init9:  call    signon                  ; display driver sign-on message
        les     di,cs:rqptr             ; restore ES:DI=request packet
        mov     word ptr es:[di.address],offset init    ; set address of
        mov     word ptr es:[di.address+2],cs           ; end of driver
        mov     byte ptr es:[di+13],1   ; driver has 1 logical unit
        mov     word ptr es:[di+20],cs  ; address of BPB array
        mov     word ptr es:[di+18],offset array
        xor     ax,ax                   ; return success status
        ret

;
; EMM initialization failed, display error message and abort
; installation of the EMSDISK device driver.
;
abort:  push    dx                      ; save error message address
        mov     ah,9                    ; fxn 9 = display string
        mov     dx,offset ermsg         ; address of error heading
        int     21h                     ; transfer to MS-DOS
        mov     ah,9                    ; fxn 9 = display string
        pop     dx                      ; address of error description
        int     21h                     ; transfer to MS-DOS
        les     di,cs:rqptr             ; restore ES:DI=request packet
        mov     word ptr es:[di.address],0     ; set break address
        mov     word ptr es:[di.address+2],cs  ; to start of driver
        mov     byte ptr es:[di+13],0   ; set logical units = 0
        xor     ax,ax                   ; but return success status
        ret

init    endp

;
; Set up total sectors, sectors per cluster, and sectors per FAT
; fields of BIOS Parameter Block according to size of RAMdisk.
;
makebpb proc    near

        mov     ax,ownedp               ; calc RAMdisk total sectors,
        mov     cx,spp                  ; update BIOS parameter block
        mul     cx
        mov     bpb+8,ax

        mov     cx,2                    ; calc sectors per cluster
makeb1: mov     ax,bpb+8                ; try this cluster size...
        mov     dx,0                    ; divide total sectors by
        div     cx                      ; sectors per cluster.
        cmp     ax,4086                 ; resulting clusters < 4087?
        jna     makeb2                  ; yes, use it
        shl     cx,1                    ; no, sec/cluster*2
        jmp     makeb1                  ; try again

makeb2: mov     byte ptr bpb+2,cl       ; sectors per cluster into BPB
        mov     dx,ax                   ; now AX = total clusters
        add     ax,ax                   ; clusters*1.5 = bytes in FAT
        add     ax,dx
        shr     ax,1
        mov     dx,0                    ; bytes in FAT/ bytes/sector
        mov     cx,ssize                ; = number of FAT sectors
        div     cx
        or      dx,dx                   ; any remainder?
        jz      makeb3                  ; no,jump
        inc     ax                      ; round up to next sector

makeb3: mov     bpb+0bh,ax              ; FAT sectors into BPB
        ret                             ; done with BPB now

makebpb endp

;
; Format RAMdisk.  First write zeros into all sectors of reserved
; area, FAT, and root directory.  Then copy phony boot record to
; boot sector, initialize medium ID byte at beginning of FAT, and
; place phony volume label in first sector of root directory.
; Returns Carry = clear if successful, Carry = set if failed.
;
format  proc    near

        mov     bx,0                    ; first clear RAMdisk area
fmt1:   cmp     bx,ownedp               ; done with all EMS pages?
        je      fmt2                    ; yes, jump
        push    bx                      ; save current page number
        mov     ax,4400h                ; fxn 4400h = map phys. page 0
        mov     dx,handle               ; EMM handle for this driver
        int     emm_int                 ; transfer to EMM
        pop     bx                      ; restore page number
        or      ah,ah                   ; if bad mapping give up
        jnz     fmt9                    ; (should never happen)
        mov     es,pframe               ; set ES:DI = EMS page
        xor     di,di
        mov     cx,psize/2              ; page length in words
        xor     ax,ax                   ; fill page with zeros
        cld
        rep stosw
        inc     bx                      ; increment page and loop
        jmp     fmt1

fmt2:                                   ; copy phony boot sector
        mov     ax,0                    ; map in logical sector 0
        call    mapsec
        jc      fmt9                    ; jump if mapping error
        mov     di,ax                   ; ES:DI = sector 0
        mov     es,pframe
        mov     si,offset bootrec       ; DS:SI = boot record
        mov     cx,br_len               ; CX = length to copy
        rep movsb                       ; transfer boot sector data
        mov     ax,1                    ; map in logical sector 1
        call    mapsec                  ; (first sector of FAT)
        jc      fmt9                    ; jump if mapping error
        mov     di,ax                   ; ES:DI = sector 1
        mov     es,pframe
        mov     al,byte ptr [bpb+0ah]   ; put media descriptor byte
        mov     es:[di],al              ; into FAT byte 0, force
        mov     word ptr es:[di+1],-1   ; bytes 1-2 to FFH
        mov     al,byte ptr [bpb+5]     ; first directory sector =
        xor     ah,ah                   ; no. of FATS * length of FAT
        mul     word ptr [bpb+0bh]      ; plus reserved sectors
        add     ax,word ptr [bpb+3]
        call    mapsec                  ; map in directory sector
        jc      fmt9                    ; jump if mapping error
        mov     di,ax                   ; copy volume label to
        mov     es,pframe               ; first sector of directory
        mov     si,offset volname
        mov     cx,vn_len
        rep movsb
        clc                             ; return CY = clear,
        ret                             ; format was successful

fmt9:   stc                             ; return CY = set,
        ret                             ; error during format

format  endp

;
; Display sign-on message, logical volume (if DOS 3.0 or later),
; amounts of installed, available, and allocated expanded memory.
;
signon  proc    near

        les     di,rqptr                ; ES:DI = request packet
        mov     al,es:[di+22]           ; get drive code from header,
        add     al,'A'                  ; convert it to ASCII, and
        mov     dcode,al                ; store into sign-on message

        mov     ax,totalp               ; format KB of EM installed
        mov     dx,16
        mul     dx                      ; pages * 16 = KB
        mov     cx,10
        mov     si,offset kbins
        call    itoa                    ; convert KB to ASCII

        mov     ax,availp               ; format KB of EM available
        mov     dx,16
        mul     dx                      ; pages * 16 = KB
        mov     cx,10
        mov     si,offset kbavail
        call    itoa                    ; convert KB to ASCII

        mov     ax,ownedp               ; format KB assigned to RAMdisk
        mov     dx,16
        mul     dx                      ; pages * 16 = KB
        mov     cx,10
        mov     si,offset kbasn
        call    itoa                    ; convert KB to ASCII

        mov     ah,9                    ; fxn 9 = display string
        mov     dx,offset ident         ; address of program name
        int     21h                     ; transfer to MS-DOS

        mov     dx,offset dos2m         ; check DOS version, if
        cmp     dosver,2                ; DOS 2 can't know drive
        je      sign1
        mov     dx,offset dos3m         ; if DOS 3 can display drive

sign1:  mov     ah,9                    ; display KB of EMS memory
        int     21h                     ; installed, available, assigned
        ret                             ; back to caller

signon  endp

;
; Convert ASCII string to 16-bit binary integer.  Overflow
; is ignored.  Conversion terminates on first illegal character.
;
; Call with:    DS:SI = address of string
;               where 'string' is in the form
;               [whitespace][sign][digits]
;
; Returns:      AX    = result
;               DS:SI = address+1 of terminator
;
atoi    proc    near                    ; ASCII to 16-bit integer

        push    bx                      ; save registers
        push    cx
        push    dx
        xor     bx,bx                   ; initialize forming answer
        xor     cx,cx                   ; initialize sign flag

atoi1:  lodsb                           ; scan off whitespace
        cmp     al,blank                ; ignore leading blanks
        je      atoi1
        cmp     al,tab                  ; ignore leading tabs
        je      atoi1

        cmp     al,'+'                  ; proceed if + sign
        je      atoi2
        cmp     al,'-'                  ; is it - sign?
        jne     atoi3                   ; no, test if numeric
        dec     cx                      ; was - sign, set flag

atoi2:  lodsb                           ; get next character

atoi3:  cmp     al,'0'                  ; is character valid?
        jb      atoi4                   ; jump if not '0' to '9'
        cmp     al,'9'
        ja      atoi4                   ; jump if not '0' to '9'
        and     ax,0fh                  ; isolate lower four bits
        xchg    bx,ax                   ; multiply answer x 10
        mov     dx,10
        mul     dx
        add     bx,ax                   ; add this digit
        jmp     atoi2                   ; convert next digit

atoi4:  mov     ax,bx                   ; result into AX
        jcxz    atoi5                   ; jump if sign flag clear
        neg     ax                      ; make result negative

atoi5:  pop     dx                      ; restore registers
        pop     cx
        pop     bx
        ret                             ; back to caller

atoi    endp

;
; Convert 16-bit binary integer to ASCII string.
;
; Call with:    AX    = 16-bit integer
;               DS:SI = buffer to receive string,
;                       must be at least 6 bytes long
;               CX    = radix
;
; Returns:      DS:SI = address of converted string
;               AX    = length of string
;
itoa    proc    near                    ; convert binary int to ASCII

        add     si,6                    ; advance to end of buffer
        push    si                      ; and save that address
        or      ax,ax                   ; test sign of 16-bit value,
        pushf                           ; and save sign on stack
        jns     itoa1                   ; jump if value was positive
        neg     ax                      ; find absolute value

itoa1:  cwd                             ; divide value by radix to
        div     cx                      ; extract next digit
        add     dl,'0'                  ; convert remainder to ASCII
        cmp     dl,'9'                  ; in case converting to hex
        jle     itoa2                   ; jump if in range 0-9
        add     dl,'A'-'9'-1            ; correct digit if in range A-F

itoa2:  dec     si                      ; back up through buffer
```

```
        mov     [si],dl                 ; store this character
        or      ax,ax                   ; value now zero?
        jnz     itoa1                   ; no, convert another digit
        popf                            ; original value negative?
        jns     itoa3                   ; no, jump
        dec     si                      ; yes,store sign into output
        mov     byte ptr [si],'-'

itoa3:  pop     ax                      ; calculate length of string
        sub     ax,si
        ret                             ; return to caller

itoa    endp

;
; Miscellaneous data and text strings used only during
; initialization, discarded afterwards to save memory.
;
ident   db      cr,lf,lf
        db      'EMSDISK Expanded Memory RAMdisk 1.1'
        db      cr,lf
        db      'Copyright (C) 1989 Ray Duncan'
        db      cr,lf,lf,eom
dos3m   db      'RAMdisk will be drive '
dcode   db      'X.'
        db      cr,lf,lf
dos2m   label   byte
kbins   db      '       KB expanded memory installed.'
        db      cr,lf
kbavail db      '       KB expanded memory available.'
        db      cr,lf
kbasn   db      '       KB assigned to RAMdisk.'
        db      cr,lf,eom

emm_name db     'EMMXXXX0',0            ; logical device name for
                                        ; expanded memory manager

ermsg   db      cr,lf
        db      'EMSDISK installation error:'
        db      cr,lf,eom

msg1    db      'expanded memory manager not found.'
        db      cr,lf,eom

msg2    db      'expanded memory not functional.'
        db      cr,lf,eom

msg3    db      'expanded memory manager error.'
```

```
        db      cr,lf,eom

msg4    db      'no expanded memory pages available.'
        db      cr,lf,eom

msg5    db      'expanded memory allocation failed.'
        db      cr,lf,eom

msg6    db      'wrong expanded memory manager version.'
        db      cr,lf,eom

msg7    db      'unable to format RAMdisk.'
        db      cr,lf,eom

volname db      'EMSDISK    '           ; phony volume label
        db      08h                     ; volume label attribute byte
        db      10 dup (0)              ; reserved
        dw      0                       ; time = 00:00:00
        dw      1441h                   ; date = February 1, 1990
        db      6 dup (0)
vn_len  equ     $-volname

_TEXT   ends

        end
```

Table 2-4: The EMS Programming Interface.

Function	EMS Version	Parameters	Results if Successful*
EMS Function 40H Get Status	3.0	AH = 40H	AH = 00H
Note: This function should only be used after an application has established that the expanded memory manager is in fact present.			
EMS Function 41H Get Page Frame Address	3.0	AH = 41H	AH = 00H BX = segment base of page frame
Note: The page frame is divided into four 16K pages. In EMS 4.0, pages may be mapped to other locations than the page frame.			

EMS Function 42H
Get Number of Pages
	3.0	AH = 42H	AH = 00H
			BX = unallocated pages
			DX = total pages

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Function	EMS Version	Parameters	Results if Successful*

EMS Function 43H
Allocate Handle and Pages
	3.0	AH = 43H
		BX = pages to allocate (must be nonzero)
			AH = 00H
			DX = EMM handle

Note: The pages allocated by this function are always 16K. Zero pages may not be allocated.			

EMS Function 44H
Map Expanded Memory Page
	3.0	AH = 44H
		AL = physical page
		BX = logical page
		DX = EMM handle
			AH = 00H

Note: In EMS 4.0, if this function is called with BX = -1, the specified physical page is unmapped.			

EMS Function 45H
Release Handle and Expanded Memory Pages
	3.0	AH = 45H
		DX = EMM handle
			AH = 00H

EMS Function 46H
Get EMS Version
	3.0	AH = 46H	AH = 00H
			AL = EMS version

Note: The version number is returned in binary coded decimal (BCD) format, with the integer portion in the upper 4 bits of AL and the fractional portion in the lower 4 bits.			

EMS Function 47H
Save Page Map
	3.0	AH = 47H
		DX = EMM handle
			AH = 00H

Note: This function saves the mapping state only for the 64K page frame defined in EMS 3.x.			

EMS Function 48H
Restore Page Map
	3.0	AH = 48H
		DX = EMM handle
			AH = 00H

Note: This function restores the mapping state only for the 64K page frame defined in EMS 3.x.			
EMS Function 49H Reserved	3.0		
EMS Function 4AH Reserved	3.0		
EMS Function 4BH Get Number of Active Handles	3.0	AH = 4BH	AH = 00H BX = number of active handles
Note: The maximum number of active handles is 255.			

EMS Function 4CH
Get Number of Pages for Handle
	3.0	AH = 4CH
		DX = EMM handle
			AH = 00H
			BX = number of pages

Note: The maximum number of pages which may be allocated to a handle is 512 in EMS 3.x and 2,048 in EMS 4.0.			

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Function	EMS Version	Parameters	Results if Successful*
EMS Function 4DH Get Pages for All Handles	3.0	AH = 4DH ES:DI = buffer address	AH = 00H BX = number of active handles *and* page information in buffer
Note: The buffer is filled in with a series of *dword* (32-bit) entries, one per active EMM handle. The first word of an entry contains the handle, and the second word contains the number of pages allocated to that handle.			

EMS Function 4EH
Subfunction 00H
Save Page Map
	3.2	AH = 4EH
		AL = 00H
		ES:DI = buffer address
			AH = 00H
			and mapping information in buffer

Note: The size of the buffer required by this function can be obtained with EMS Function 4EH Subfunction 03H.			

EMS Function 4EH
Subfunction 01H
Restore Page Map
	3.2	AH = 4EH
		AL = 01H
		DS:SI = buffer address
			AH = 00H

Note: The mapping information in the buffer must be prepared by a previous call to EMS Function 4EH Subfunction 00H or 02H.			

EMS Function 4EH
Subfunction 02H
Save and Restore Page Map
	3.2	AH = 4EH
		AL = 02H
		DS:SI = buffer containing mapping information
		ES:DI = buffer to receive mapping information
			AH = 00H
			and mapping information in buffer pointed to by ES:DI

Note: The mapping information in the buffer pointed to by DS:SI must be prepared by a previous call to EMS Function 4EH Subfunction 00H or 02H. The size of the buffers required by this function can be obtained with EMS Function 4EH Subfunction 03H.			

EMS Function 4EH
Subfunction 03H
Get Size of Page Map Information
	3.2	AH = 4EH
		AL = 03H
			AH = 00H
			AL = buffer size (bytes)

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Function	EMS Version	Parameters	Results if Successful*

EMS Function 4FH
Subfunction 00H
Save Partial Page Map
	4.0	AH = 4FH
		AL = 00H
		DS:SI = map list
		ES:DI = buffer to receive mapping state
			AH = 00H
			and buffer pointed to by ES:DI filled in with mapping information

Note: The map list contains the number of mappable segments in the first word, followed by the segment addresses of the mappable memory regions (one segment per word). EMS Function 4FH Subfunction 02H can be called to obtain the size of the buffer to receive the mapping information.			

EMS Function 4FH
Subfunction 01H
Restore Partial Page Map
	4.0	AH = 4FH
		AL = 01H
		DS:SI = address of buffer containing mapping information
			AH = 00H

Note: The buffer containing the mapping information must be prepared by a previous call to EMS Function 4FH Subfunction 00H.			

EMS Function 4FH
Subfunction 02H
Get Size of Partial Page Map Information
	4.0	AH = 4FH
		AL = 02H
		BX = number of pages
			AH = 00H
			AL = buffer size (bytes)

EMS Function 50H
Subfunction 00H
Map Multiple Pages by Number
	4.0	AH = 50H
		AL = 00H
		CX = number of pages
		DX = EMM handle
		DS:SI = address of buffer containing mapping information
			AH = 00H

Note: The buffer contains a series of *dword* entries which control the pages to be mapped. The first word of each entry contains the logical page number, and the second word contains the physical page number. If the logical page is -1 the physical page is unmapped.			

EMS Function 50H
Subfunction 01H
Map Multiple Pages by Address
	4.0	AH = 50H
		AL = 01H
		CX = number of pages
		DX = EMM handle
		DS:SI = address of buffer containing mapping information
			AH = 00H

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Function	EMS Version	Parameters	Results if Successful*
Note: The buffer contains a series of *dword* entries which control the pages to be mapped. The first word of each entry contains the logical page number, and the second word contains the physical segment address. If the logical page is -1, the physical page is unmapped.			

EMS Function 51H
Reallocate Pages for Handle
	4.0	AH = 51H
		BX = new number of pages
		DX = EMM handle
			AH = 00H
			BX = pages owned by handle

Note: If the requested number of pages is zero, the handle is still active and its allocation can be changed again at a later time.			

EMS Function 52H
Subfunction 00H
Get Handle Attribute
	4.0	AH = 52H
		AL = 00H
		DX = EMM handle
			AH = 00H
			AL = attribute
			0 = volatile
			1 = nonvolatile

Note: A non-volatile memory handle and the contents of the expanded memory pages which are allocated to it are maintained across a system restart using Ctrl-Alt-Del.			

EMS Function 52H
Subfunction 01H
Set Handle Attribute
	4.0	AH = 52H
		AL = 01H
		BL = attribute
		0 = volatile
		1 = nonvolatile
		DX = EMM handle
			AH = 00H

Note: If the system does not support non-volatile handles, an error is returned.			

EMS Function 52H
Subfunction 02H
Get Attribute Capability
	4.0	AH = 52H
		AL = 02H
			AH = 00H
			AL = handle attribute capability
			0 = volatile only
			1 = volatile and non-volatile

EMS Function 53H
Subfunction 00H
Get Handle Name
	4.0	AH = 53H
		AL = 00H
		DX = EMM handle
		ES:DI = buffer address
			AH = 00H
			and buffer contains 8-byte handle name

Note: A handle's name is initialized to 8 zero bytes when it is allocated or deallocated.			

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Function	EMS Version	Parameters	Results if Successful*

EMS Function 53H
Subfunction 01H
Set Handle Name
	4.0	AH = 53H
		AL = 01H
		DX = EMM handle
		DS:SI = address of 8-byte handle name
			AH = 00H

Note: The bytes in a handle name need not be ASCII characters. The default name for a handle is 8 zero bytes. The name of a non-volatile handle will be preserved across a warm boot.

EMS Function 54H
Subfunction 00H
Get All Handle Names
	4.0	AH = 54H
		AL = 00H
		ES:DI = buffer address
			AH = 00H
			AL = number of active handles
			and handle name information in buffer

Note: The buffer is filled with a series of 10-byte entries. The first two bytes of an entry contain an EMM handle, and the next eight bytes contain the handle's name. The maximum size of the returned information is 2,550 bytes.

EMS Function 54H
Subfunction 01H
Search for Handle Name
	4.0	AH = 54H
		AL = 01H
		DS:SI = address of 8-byte handle name
			AH = 00H
			DX = EMM handle

EMS Function 54H
Subfunction 02H
Get Total Handles
	4.0	AH = 54H
		AL = 02H
			AH = 00H
			BX = number of handles

EMS Function 55H
Subfunctions 00H and 01H
Map Pages and Jump
	4.0	AH = 55H
		AL = 0 *to map by physical page numbers,* 1 *to map by physical page segments*
		DX = EMM handle
		DS:SI = buffer address
			AH = 00H

Note: The buffer pointed to by DS:SI is formatted as:

dword	far pointer to jump target
byte	number of pages to map before jump
dword	far pointer to map list

The map list consists of *dword* entries; one per page to be mapped. The first word of an entry contains the logical page number, and the second word contains a physical page number or segment (depending on the value in AL).

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Function	EMS Version	Parameters	Results if Successful*

EMS Function 56H
Subfunctions 00H and 01H
Map Pages and Call
	4.0	AH = 56H
		AL = 0 *to map by physical page numbers,* 1 *to map by physical page segments*
		DX = EMM handle
		DS:SI = buffer address
			AH = 00H

Note: The buffer pointed to by DS:SI is formatted as:

dword	far pointer to call target
byte	number of pages to map before call
dword	far pointer to call map list
byte	number of pages to map before return
dword	far pointer to return map list
8 bytes	reserved

Both map lists consist of *dword* entries; one per page to be mapped. The first word of an entry contains the logical page number, and the second word contains a physical page number or segment (depending on the value in AL).

Function	EMS Version	Parameters	Results if Successful*

EMS Function 56H
Subfunction 02H
Get Stack Space Required for Map Page and Call
	4.0	AH = 56H
		AL = 02H
			AH = 00H
			BX = stack space required (bytes)

EMS Function 57H
Subfunction 00H
Move Memory Region
	4.0	AH = 57H
		AL = 00H
		DS:SI = buffer address
			AH = 00H

Note: The buffer pointed to by DS:SI controls the move operation and is formatted as:

dword	region length in bytes
byte	source memory type (0 = conventional, 1 = expanded)
word	source memory handle
word	source memory offset
word	source memory segment or logical page number
byte	destination memory type (0 = conventional, 1 = expanded)
word	destination memory handle
word	destination memory offset
word	destination memory segment or logical page number

The maximum length of a move is 1 megabyte. If the length exceeds a single page, consecutive pages supply or receive the data. Overlapping addresses are handled correctly.

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Function	EMS Version	Parameters	Results if Successful*

EMS Function 57H
Subfunction 01H
Exchange Memory Regions
	4.0	AH = 57H
		AL = 01H
		DS:SI = buffer address
			AH = 00H

Note: The format of the buffer controlling the exchange operation is the same as for EMS Function 57H Subfunction 00H. The maximum length of an exchange is 1 megabyte. Consecutive pages are used as required. Source and destination addresses may not overlap.			

EMS Function 58H
Subfunction 00H
Get Addresses of Mappable Pages
	4.0	AH = 58H
		AL = 00H
		ES:DI = buffer address
			AH = 00H
			CX = number of entries in buffer
			and address information placed in buffer

Note: The returned information in the buffer consists of *dword* entries, one per mappable page. The first word of an entry contains the page's segment base address, and the second contains its physical page number. The entries are sorted in order of ascending segment addresses.			

EMS Function 58H
Subfunction 01H
Get Number of Mappable Pages
	4.0	AH = 58H
		AL = 01H
			AH = 00H
			CX = number of mappable pages

EMS Function 59H
Subfunction 00H
Get Hardware Configuration
	4.0	AH = 59H
		AL = 00H
		ES:DI = buffer address
			AH = 00H
			and hardware configuration information in buffer

Note: The format of the information returned in the buffer is: *word* size of raw expanded memory pages (paragraphs) *word* number of alternate register sets *word* size of context save area (bytes) *word* number of register sets assignable to DMA channels *word* DMA operation type (0 = DMA can be used with alternate register sets, 1 = only one DMA register set available)			

EMS Function 59H
Subfunction 01H
Get Number of Raw Pages
	4.0	AH = 59H
		AL = 01H
			AH = 00H
			BX = number of free raw pages
			DX = total raw pages

Note: Raw memory pages may have a size other than 16K.			

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Function	EMS Version	Parameters	Results if Successful*

EMS Function 5AH
Subfunction 00H
Allocate Handle and Standard Pages
	4.0	AH = 5AH
		AL = 00H
		BX = number of 16K pages
			AH = 00H
			DX = EMM handle

Note: Allocation of zero pages with this function is not an error.			

EMS Function 5AH
Subfunction 01H
Allocate Handle and Raw Pages
	4.0	AH = 5AH
		AL = 01H
		BX = number of raw pages
			AH = 00H
			DX = EMM handle

Note: Raw memory pages may have a size other than 16K. Allocation of zero pages is not an error.			

EMS Function 5BH
Subfunction 00H
Get Alternate Map Registers
	4.0	AH = 5BH
		AL = 00H
			AH = 00H
			BL = current alternate register set, or zero if alternate set not active
			ES:DI = address of alternate map register set save area (if BL = 0)

Note: The address of the save area must be specified in a previous call to EMS Function 5BH Subfunction 01H, and the save area initialized with a previous call to EMS Function 4EH Subfunction 00H.			

EMS Function 5BH
Subfunction 01H
Set Alternate Map Registers
	4.0	AH = 5BH
		AL = 01H
		BL = alternate map register set number, or zero
		ES:DI = address of map register context save area (if BL = 0)
			AH = 00H

Note: The buffer address specified in this call is returned by subsequent calls to EMS Function 5BH Subfunction 00H. The save area must be initialized by a previous call to EMS Function 4EH Subfunction 00H.			

EMS Function 5BH
Subfunction 02H
Get Size of Alternate Map Register Save Area
	4.0	AH = 5BH
		AL = 02H
			AH = 00H
			DX = size of buffer required (bytes)

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Function	EMS Version	Parameters	Results if Successful*

EMS Function 5BH
Subfunction 03H
Allocate Alternate Map Register Set
	4.0	AH = 5BH
		AL = 03H
			AH = 00H
			BL = alternate map register set number, or zero if no alternates available

Note: The contents of the currently active map registers are copied into the newly allocated alternate map registers.			

EMS Function 5BH
Subfunction 04H
Deallocate Alternate Map Register Set
	4.0	AH = 5BH
		AL = 04H
		BL = alternate map register set number
			AH = 00H

Note: The current alternate map register set cannot be deallocated.			

EMS Function 5BH
Subfunction 05H
Allocate DMA Register Set
	4.0	AH = 5BH
		AL = 05H
			AH = 00H
			BL = DMA register set number, or 0 if none available

EMS Function 5BH
Subfunction 06H
Enable DMA on Alternate Map Register Set
	4.0	AH = 5BH
		AL = 06H
		BL = alternate map register set number
		DL = DMA channel number
			AH = 00H

Note: If a DMA channel is not assigned to a specific register set, DMA for that channel will be mapped through the current register set.			

EMS Function 5BH
Subfunction 07H
Disable DMA on Alternate Map Register Set
	4.0	AH = 5BH
		AL = 07H
		BL = alternate map register set number
			AH = 00H

EMS Function 5BH
Subfunction 08H
Deallocate DMA Register Set
	4.0	AH = 5BH
		AL = 08H
		BL = DMA register set number
			AH = 00H

EMS Function 5CH Prepare Expanded Memory Manager for Warm Boot	4.0	AH = 5CH	AH = 00H

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Function	EMS Version	Parameters	Results if Successful*
Note: This function affects the current mapping context, the alternate register set in use, and any other hardware dependencies that would ordinarily be initialized when the system is reset.			

EMS Function 5DH
Subfunction 00H
Enable EMM Operating
System Functions
	4.0	AH = 5DH
		AL = 00H
		BX:CX = access key (if not first call)
			AH = 00H
			BX:CX = access key (if first call)

Note: Enables EMS Functions 59H, 5BH, and 5DH (this is the default condition). An access key is returned on the first call to either Subfunction 00H or 01H of EMS Function 5DH. This key must be used in subsequent calls to either subfunction.			

EMS Function 5DH
Subfunction 01H
Disable EMM Operating
System Functions
	4.0	AH = 5DH
		AL = 01H
		BX:CX = access key (if not first call)
			AH = 00H
			BX:CX = access key (if first call)

Note: Disables EMS Functions 59H, 5BH, and 5DH.			

EMS Function 5DH
Subfunction 02H
Release Access Key
	4.0	AH = 5DH
		AL = 02H
		BX:CX = access key
			AH = 00H

Note: A new access key will be returned by the next call to EMS Function 5DH Subfunction 00H or 01H.			

*Failure of an EMS Function is indicated by return of a nonzero error code in register AH. A list of these error codes may be found in Table 2-5.

Table 2-5: Expanded Memory Manager standardized error codes.
The error codes 90H and above are only supported in EMS version 4.0.

Error Code	Meaning
80H	Internal error in expanded memory manager software (may indicated corrupted memory image of driver)
81H	Malfunction in expanded memory hardware
82H	Memory manager busy
83H	Invalid handle
84H	Function not defined
85H	Handles exhausted
86H	Error in save or restore of mapping context
87H	Allocation request specified more pages than are physically available in system; no pages were allocated
88H	Allocation request specified more pages than are currently available; no pages were allocated
89H	Zero pages cannot be allocated

Error Code	Meaning
8AH	Requested logical page is outside range of pages owned by handle
8BH	Illegal physical page number in mapping request
8CH	Page mapping hardware-state save area is full
8DH	Mapping context save failed; save area already contains context associated with specified handle
8EH	Mapping context restore failed; save area does not contain context for specified handle
8FH	Subfunction parameter not defined
90H	Attribute type not defined
91H	Feature not supported
92H	Source and destination memory regions have same handle and overlap; requested move was performed, but part of the source region was overwritten
93H	Specified length for source or destination memory region is longer than actual allocated length
94H	Conventional memory region and expanded memory region overlap
95H	Specified offset is outside logical page
96H	Region length exceeds 1 megabyte
97H	Source and destination memory regions have same handle and overlap; exchange cannot be performed
98H	Memory source and destination types are undefined
99H	Error code currently unused
9AH	Alternate map or DMA register sets are supported, but specified alternate register set is not supported
9BH	Alternate map or DMA register sets are supported, but all alternate register sets are currently allocated
9CH	Alternate map or DMA register sets are not supported, and specified alternate register set is not zero
9DH	Alternate map or DMA register sets are supported, but the alternate register set specified is not defined or not allocated
9EH	Dedicated DMA channels not supported
9FH	Dedicated DMA channels are supported, but specified DMA channel is not supported
A0H	No handle found for specified name
A1H	Handle with same name already exists
A3H	Invalid pointer passed to function, or contents of source array corrupted
A4H	Access to function denied by operating system

Chapter 3

Extended Memory and the XMS

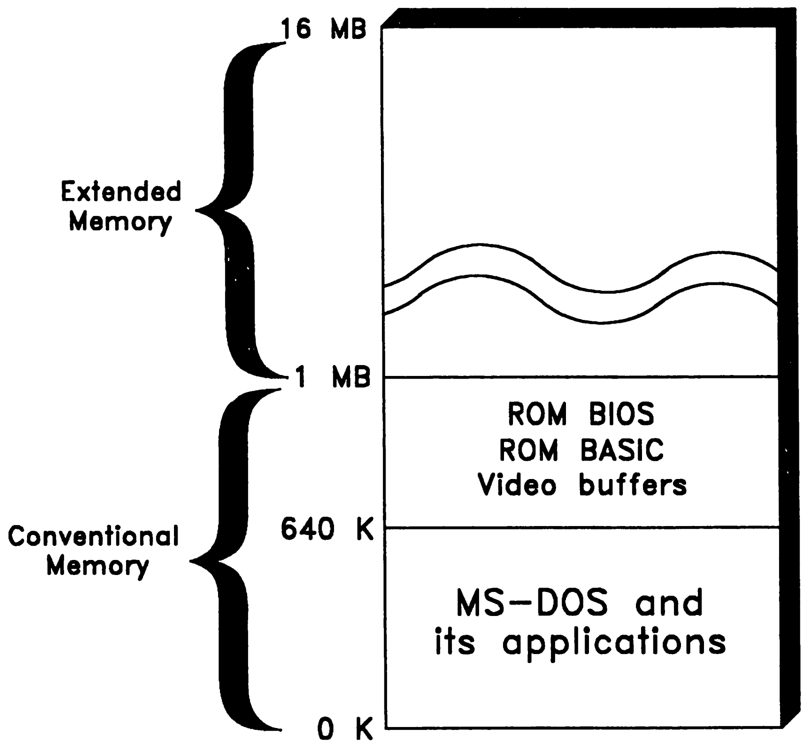
Ray Duncan

Extended memory is the term for RAM storage at addresses above the 1-megabyte boundary on 80286-, 80386-, and 80486-based PCs. This distinguishes such memory from *conventional memory,* which is at addresses below 1 megabyte, or *expanded memory,* which is essentially bank-switched memory divided into pages that can be mapped into the conventional memory address space (expanded memory is discussed in Chapter 2). A sketch of the relationship between conventional memory and extended memory is shown in Figure 3-1.

If you own a PC/AT clone of almost any brand, you probably have at least a small amount of extended memory in your system. These days, such clones typically arrive with 1 megabyte or more of RAM installed on the motherboard, of which 512K or 640K starts at address 0, and the remainder begins at 1 megabyte. In addition, if you have purchased an add-in memory board for an AT-class machine, that board can probably be configured either as extended memory, expanded memory, or a combination of both.

Thus, extended memory is a readily available resource, and protected-mode operating systems such as OS/2 and UNIX can effectively use all the extended memory you can plug into your machine for execution of programs and storage of data. MS-DOS and its client programs, on the other hand, can gain access to this memory only with great difficulty. Why? The reason is neither complicated nor obscure. It is because MS-DOS runs on 80286/386/486 CPUs in real mode—a sort of 8086/88 emulation mode—which has important implications for the way addresses are generated.

Figure 3-1: Relationships between conventional and extended memory.



Programmers think in terms of segments, selectors, and offsets, but the CPU views memory as a simple, linearly addressed array of bytes. In real mode, the CPU selects a particular memory location by shifting the contents of a segment register left four bits and adding it to a 16-bit offset, forming a 20-bit physical address. But extended memory lies (by definition) above the 1-megabyte boundary (100000H), so all physical addresses that correspond to extended memory have at least 21 significant bits. In other words, real-mode programs can't "see" extended memory because they simply can't generate the appropriate addresses.

There are ways around this seemingly impenetrable addressing barrier, however, as we all know from our own daily experience. We've all got RAMdisks, disk caches, print spoolers, and TSRs that ostensibly run in real mode but are able to exploit extended memory when it is present. The eXtended Memory Specification (XMS), which was released in 1988 as a collaborative effort of Microsoft, Intel, Lotus, and AST Research, was designed to bring all such programs into harmony: it defines a software interface for extended memory access comparable to the role of the LIM EMS for expanded memory.

Unfortunately, it will not suffice to simply describe the XMS and be done with it, as we could safely do for EMS in Chapter 2. By the time the XMS appeared, 80286-based PCs had been on the market for four years, and other methods of extended memory access and management had already evolved and were in common use. Today's software developer who wishes to write programs that are extended memory-aware, and that will be compatible with the widest possible range of other software, faces a rather complex situation, as we shall see in this chapter.

Reaching Extended Memory in Real Mode

The first thing to understand about using extended memory is that there is no free lunch: a program *does* (with two bizarre exceptions, to be explained later in this chapter) need to be running in protected mode in order to read and write memory locations above the 1-megabyte boundary. And moving safely from real mode to protected mode and back again is a nontrivial chore.

The first step, getting into protected mode from real mode, is not all that difficult. Simply set the PE (protect enable) bit in the CPU's machine status word (known as MSW on the 80286, CR0 on the 80386 and 80486). Unless the other required housekeeping has been done, though, your program will just crash immediately. As we saw in Chapter 1, certain data structures and CPU registers must be initialized for protected-mode execution that have no meaning in real mode. For example, your program must set up a global descriptor table (GDT) that controls protected-mode memory mapping, segment types, and access rights; load the address of the table into the CPU's GDT pointer register; and finally, load all segment registers with valid selectors that refer to the GDT.

Assuming that your program manages to enter protected mode properly, and read or write the data in extended memory that it is interested in, it must then return to real mode to continue its main line of execution. After all, your program needs to be able to invoke MS-DOS to read or write files and interact with the user, but MS-DOS will be quite confused if your program calls it in protected mode.

Faced with this challenge, your first inclination might be to haul down your handy *Intel 80286 Programmer's Reference* and look up the machine instruction that switches the CPU from protected mode to real mode. Surprisingly enough, there is no such instruction. When the 80286 was designed, the Intel engineers never dreamed that somebody would ever want to make a transition from the clearly superior protected mode back to dull old real mode! Luckily, there is an escape hatch, however undesirable it may sound: if the CPU is halted and restarted, it restarts in real mode.

On 80286-based PC/AT class machines, the actual technique used by the ROM BIOS (and hence by VDISK and most other extended-memory-aware programs) to return to real mode is as follows: a "magic" value is stored into a reserved memory location, the contents of the stack and general registers are saved in other reserved memory locations, a special command is sent to the keyboard controller, and the CPU is halted. The keyboard controller, in its own good time, recognizes the command and responds with a signal that resets the CPU.

After the reset by the keyboard controller, the CPU begins execution at `FFFF:0000H`, as usual, and enters the ROM BIOS Power-Up-Self-Test (POST) sequence. The POST checks for the "magic" value that was saved in RAM earlier, recognizes that the machine is waking up from an intentional halt, restores the stack and registers, and returns control to the previously executing program rather than continuing with the ROM bootstrap. The turnaround time on this process can be on the order of several milliseconds, and it has been aptly characterized by Microsoft's Gordon Letwin as "turning off the car to change gears."

Things aren't quite so bad on 80286-based PS/2 or 80386/486-based machines. 80286-based PS/2s have special hardware support that allows a faster reset cycle (though the CPU still needs to be halted to accomplish it). 80386/486-based machines, on the other hand, can accomplish the switch back to real mode by simply clearing the PE bit in CR0, and don't need to halt the CPU at all. This is because by the time the 80386 was being designed, it was becoming obvious that MS-DOS and the programs that run under it weren't going to disappear.

The ROM BIOS Extended Memory Functions

Luckily, even from the earliest days of the PC/AT, MS-DOS programmers who wish to use extended memory for data storage have never needed to worry too much about the details of protected-mode programming and mode transitions. The PC/AT ROM BIOS provides two functions that give access to extended memory in a hardware-independent manner: `Int 15H` Function `87H`, which copies a block of data from any location in conventional or extended memory to any other location, and `Int 15H` Function `88H`, which returns the amount of extended memory installed in the system. The parameters and results of these two functions are outlined below:

Int 15H Function 88H—Get Extended Memory Size

Call with:
```
AH   = 88H
```
Returns:
```
AX   = amount of extended memory (in KB)
```

Int 15H Function 87H—Move Extended Memory Block

Call with:
```
AH            = 87H
CX            = number of words to move
ES:SI         = segment:offset of global descriptor table
```
Returns:

If function successful
```
Carry flag    = clear
AH            = 00H
```

If function unsuccessful
```
Carry flag    = set
AH            = status
                01H     if RAM parity error
                02H     if exception interrupt error
                03H     if gate address line 20 failed
```

When `Int 15H` Function `87` is called, registers ES:SI point to a partially filled-in global descriptor table (GDT), with room for six descriptors (see Figure 3-2). The first descriptor is a dummy and corresponds to a null selector in the range `0000-0003H`. Null selectors get special treatment from the hardware in protected mode; they are safe values that you can always load into a segment register as long as you don't try to address anything with them.

Figure 3-2: The descriptor table used by ROM BIOS Int 15H Function 87H.

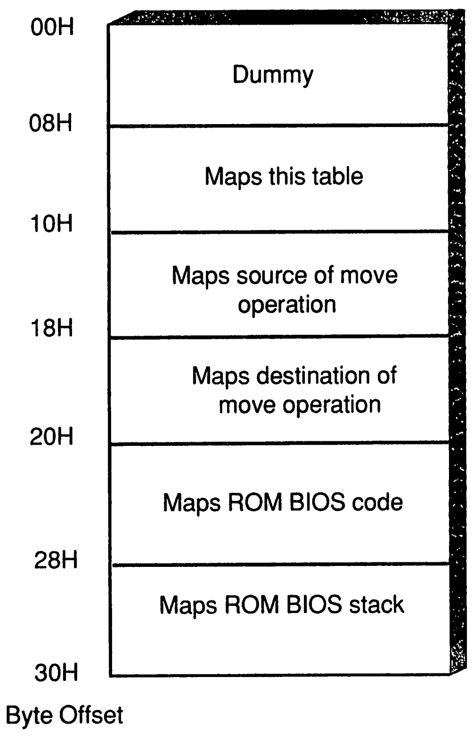


Table 3-1: The portions of the global descriptor table for Int 15H Function 87H that must be initialized by the calling program.

Byte(s)	Contents
00H-0FH	reserved (should be 0)
10H-11H	segment length in bytes (2*CX-1 or greater)
12H-14H	24-bit linear source address
15H	access rights byte (always 93H)
16H-17H	reserved (should be 0)
18H-19H	segment length in bytes (2*CX-1 or greater)
1AH-1CH	24-bit linear destination address
1DH	access rights byte (always 93H)
1EH-2FH	reserved (should be 0)

Two of the descriptors supply the source and destination addresses of the memory block that the program is asking the ROM BIOS to move on its behalf.

The descriptors must be initialized with base addresses, an appropriate length, and an "access rights" byte of 93H. The remaining three descriptors are used by the ROM BIOS to provide addressability to its own code, data, and stack while it is executing in protected mode. The calling program initializes these to zero, and the ROM BIOS takes care of the remaining necessary initialization of the table before it switches the CPU into protected mode.

The most important thing you need to notice about the descriptor table is that the addresses you place in it are 24-bit linear byte addresses—numbers from 000000H to FFFFFFH—rather than the more familiar segment:offset pairs. As we have already said, to convert the latter into the former, you merely shift the segment left 4 bits and then add the offset. The three bytes of a linear address are stored in their natural order, with the least significant byte at the lowest address.

The easiest way to cope with extended memory in an application program is to encapsulate the Int 15H Function 87H function calls inside MASM subroutines with more sensible parameters. The source file EXTMEM.ASM, shown below, contains two such routines for use with small model C programs: GETXM and PUTXM. These procedures are called with source and destination addresses and a length in bytes. The conventional memory address is assumed to be a normal far pointer (segment and offset), while the extended memory address is a linear, physical address.

```
; EXTMEM.ASM --- Routines to transfer data between
;                conventional and extended memory.
;                For use with small model C programs.
; Copyright (C) 1989 Ray Duncan
;
; Assemble with: MASM /Zi /Mx EXTMEM;

DGROUP  group   _DATA

_DATA   segment word public 'DATA'

gdt     db      30h dup (0)     ; global descriptor table

_DATA   ends


_TEXT   segment word public 'CODE'

        assume  cs:_TEXT,ds:DGROUP

args    equ     [bp+4]          ; offset of arguments, small model
```

```
source  equ     word ptr args
dest    equ     word ptr source+4
len     equ     word ptr dest+4

;
; GETXM copies data from extended memory to conventional memory.
;
; status = getxm(unsigned long source, void far *dest, unsigned len)
;
; Status is zero if move successful, nonzero if move failed:
; 1 = parity error, 2 = exception interrupt error, 3 = gate A20 failed
;
        public  _getxm
_getxm  proc    near

        push    bp              ; set up stack frame
        mov     bp,sp
        push    si              ; protect register variables
        push    di

                                ; DS: SI points to GDT
        mov     si,offset DGROUP:gdt

                                ; store access rights bytes
        mov     byte ptr [si+15h],93h
        mov     byte ptr [si+1dh],93h

        mov     ax,source       ; store source address
        mov     [si+12h],ax     ; into descriptor
        mov     ax,source+2
        mov     [si+14h],al

        mov     ax,dest+2       ; destination segment * 16
        mov     dx,16
        mul     dx
        add     ax,dest         ; + offset -> linear address
        adc     dx,0
        mov     [si+1ah],ax     ; store destination address
        mov     [si+1ch],dl     ; into descriptor

        mov     cx,len          ; store length into source
        mov     [si+10h],cx     ; and destination descriptors
        mov     [si+18h],cx

        shr     cx,1            ; convert length to words
        mov     ah,87h          ; Int 15H Fxn 87h = block move
        int     15h             ; transfer to ROM BIOS
        mov     al,ah           ; form status in AX
        cbw

        pop     di              ; restore registers
        pop     si
        pop     bp
        ret                     ; back to caller

_getxm  endp

;
; PUTXM copies data from conventional memory to extended memory.
;
; status = putxm(void far *source, unsigned long dest, unsigned len)
;
; Status is zero if move successful, nonzero if move failed:
; 1 = parity error, 2 = exception interrupt error, 3 = gate A20 failed
;
        public  _putxm
_putxm  proc    near

        push    bp              ; set up stack frame
        mov     bp,sp
        push    si              ; protect register variables
        push    di

                                ; DS: SI points to GDT
        mov     si,offset DGROUP:gdt

                                ; store access rights bytes
        mov     byte ptr [si+15h],93h
        mov     byte ptr [si+1dh],93h

        mov     ax,dest         ; store destination address
        mov     [si+1ah],ax     ; into descriptor
        mov     ax,dest+2
        mov     [si+1ch],al

        mov     ax,source+2     ; source segment * 16
        mov     dx,16
        mul     dx
        add     ax,source       ; + offset -> linear address
        adc     dx,0
        mov     [si+12h],ax     ; store source address
        mov     [si+14h],dl     ; into descriptor

        mov     cx,len          ; store length into source
        mov     [si+10h],cx     ; and destination descriptors
        mov     [si+18h],cx

        shr     cx,1                ; convert length to words
        mov     ah,87h              ; Int 15H Fxn 87h = block move
        int     15h                 ; transfer to ROM BIOS

        mov     al,ah               ; form status in AX
        cbw

        pop     di                  ; restore registers
        pop     si
        pop     bp
        ret                         ; back to caller

_putxm  endp

_TEXT   ends

        end
```

GETXM and PUTXM do all the necessary housekeeping required by the ROM BIOS, converting addresses as necessary and placing the addresses, lengths, and access right bytes into the descriptor table. Both routines return a false flag if the move was successful, or a true flag if it failed. In the latter case, the value of the flag is 1 if there was a memory parity error, 2 if an interrupt exception occurred, or 3 if extended memory could not be accessed due to a problem with the A20 address line.

Primitive Extended Memory Management

You've probably noticed the major flaw in the extended memory functions supported by the ROM BIOS: while they let you access any location in extended memory quite freely, they do not make any attempt to arbitrate between two or more programs or drivers that are using extended memory at the same time. For example, if both an application program and a RAMdisk use the ROM BIOS functions to put data in the same area of extended memory, no error is returned to either program, but the data of one or both programs may be destroyed.

Since neither IBM nor Microsoft came up with any standard scheme for the cooperative use of extended memory by DOS programs during the first few years of the PC/AT's existence, third-party software developers were left to their own devices. Eventually, almost all of them settled on one of two methods for extended memory management, which we may call the "VDISK method" and the "Int 15H method."

VDISK.SYS is a fairly conventional RAMdisk installable device driver that IBM has been supplying with PC-DOS since version 3.0. From the beginning, VDISK was capable of using either conventional or extended memory to create a virtual disk drive and, in the most recent versions, can make use of expanded memory as well. The source code for VDISK has always been included in the PC-DOS retail package, so it has (for better or worse) become a model for the implementation of many other companies' RAMdisks.

When VDISK is loaded, it allocates extended memory to itself from the 1-megabyte boundary, upwards, and saves information about the amount of extended memory it is using in two places: in a data structure located in conventional memory and found via the `Int 19H` vector, and in a data structure located in extended memory at the 1-megabyte boundary. If additional copies of VDISK are loaded (to create additional logical RAMdisk drives), they look at each of these areas to determine the amount and location of extended memory still available, then update them to reflect any additional extended memory they have reserved for their own use.

Applications which adopt the VDISK method of extended memory management merely need to inspect and update the `Int 19H` and extended memory indicators in the same manner as VDISK itself. Unfortunately, in actual practice, some applications update only the `Int 19H` area and some update only the extended memory area. This means that if you adopt the VDISK technique in your own applications, you must program very defensively and check both areas. If the two indicators differ, you must assume that the lesser amount of extended memory is available, then update both allocation signatures to be correct and consistent for any programs that are loaded after yours.

The `Int 15H` method of extended memory management is much less complicated. The application calls `Int 15H` Function `88H` first to find out how much extended memory is available, then "hooks" the `Int 15H` vector to intercept calls by other programs. When the program sees a subsequent call to `Int 15H` Function `88H`, it returns a reduced value that reflects the amount of extended memory it is using (passing all other `Int 15H` calls onward to the original owner of the interrupt vector). In this way, the program can deceive subsequently loaded applications into believing that the extended memory it is using does not exist.

In summary, the VDISK method allows extended memory to be allocated upward from the 1-megabyte boundary, and the `Int 15H` method allows extended memory to be allocated downward from the top. Both management methods are in common use, so you must take both into account in your own programs if you intend to use extended memory at all.

The VDISK Indicators

Now we can examine the specific details of how the VDISK memory management approach works.

VDISK takes over the `Int 19H` vector, which normally contains the address of the ROM BIOS routine to reboot the system, and points it to an `Int 19H` handler within itself. This new handler does nothing more than transfer control to the original handler, so its presence does not affect the system's operation at all. However, a program can fetch the segment portion of the `Int 19H` vector, assume that it points to the beginning of a VDISK driver if one is loaded, and use it to determine whether a VDISK driver is, in fact, present. If a VDISK driver is loaded, its name and the address of the first free (unallocated) extended memory can be found at fixed offsets from its base.

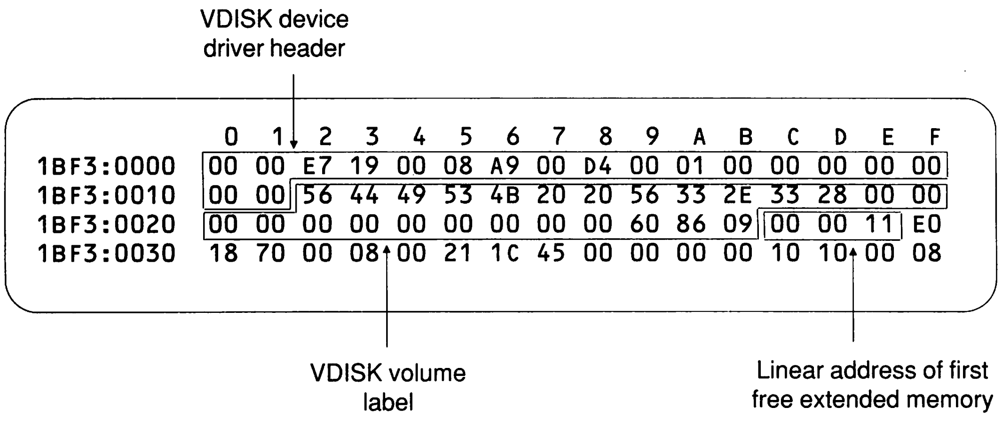
The exact memory addresses to be inspected may vary from one version of VDISK and PC-DOS to another, but you can extract the necessary information from the VDISK.ASM source file that is included on the IBM PC-DOS distribution disks. As an example, suppose we placed the line:

```
    DEVICE=VDISK.SYS /E
```

in the CONFIG.SYS file for a PC-DOS 3.3 system and rebooted. (The /E switch directs VDISK to use extended memory.) During system initialization VDISK would display a message advising that it had created a 64K RAMdisk (the default size) on logical drive F. We then inspect the `Int 19H` vector, and find that it contains the address `1BF3:008EH`. Figure 3-3 contains a hex dump of addresses `1BF3:0000H` through `1BF3:003FH`—the first 64 bytes of the VDISK driver.

Bytes `00H` through `11H` are the VDISK device driver header, which contains information about the driver's entry points, capabilities, and other information of interest to the MS-DOS kernel. In this example, bytes `12H` through `2BH` are the initial portion of a volume label that VDISK places in the root directory of its RAMDISK. As you can see, the label contains the string "VDISK," and the PC-DOS version number. Finally, bytes `2CH` through `2EH` contain the linear address of the first free byte of extended memory: `110000H` in this example (1MB + 64K, since VDISK is using the 64K starting at 1 megabyte).

Figure 3-3: The first 64 bytes of the VDISK device driver for PC-DOS 3.3.


VDISK device driver header
VDISK volume label
Linear address of first free extended memory

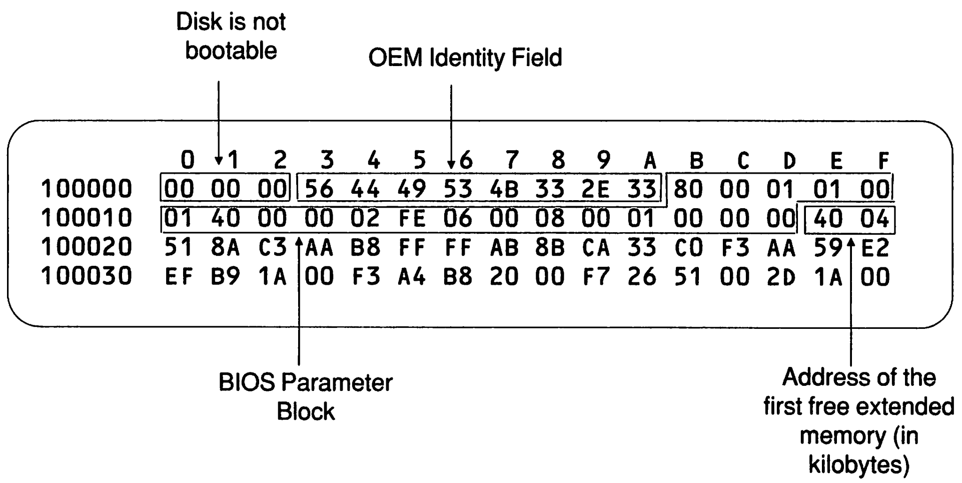
```
            0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
1BF3:0000  00 00 E7 19 00 08 A9 00 D4 00 01 00 00 00 00 00
1BF3:0010  00 00 56 44 49 53 4B 20 20 56 33 2E 33 28 00 00
1BF3:0020  00 00 00 00 00 00 00 00 00 60 86 09 00 00 11 E0
1BF3:0030  18 70 00 08 00 21 1C 45 00 00 00 00 10 10 00 08
```

Now let's take a look at the VDISK allocation information stored in extended memory. Figure 3-4 contains a hex dump of addresses 100000H through 10003FH, in other words, the first 64 bytes at the 1-megabyte boundary. This memory is part of the first logical sector of VDISK's RAMdisk storage, so VDISK makes it look like the boot sector of a normal MS-DOS block device. Offsets 00H–02H contain zero to show that the disk is not bootable, bytes 03H–0AH are the "OEM identity field" and contain the string "VDISK3.3," and bytes 0BH–1DH are the "BIOS Parameter Block" (BPB) from which MS-DOS can calculate the locations of the FAT, root directory, and so on.

The two bytes at offset 01EH and 1FH are the ones we are particularly interested in here. By studying the source code for VDISK, we find that these two bytes are treated as a WORD field, and contain the address of the first free extended memory in kilobytes. In this particular case, the word contains 0440H (1088), which is again 1MB (1024K) + 64K.

The responsibilities of a program that wants to use the VDISK method for extended memory management are now more clear. It must first find the total amount of extended memory available by calling Int 15H Function 88H (this protects it against programs that use the Int 15H management method). It must then inspect the Int 19H vector to determine whether the vector points to the base of a previously loaded VDISK driver.

Figure 3-4: The first 64 bytes of extended memory when VDISK is loaded.



```
         0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
100000  00 00 00 56 44 49 53 4B 33 2E 33 80 00 01 01 00
100010  01 40 00 00 02 FE 06 00 08 00 01 00 00 00 40 04
100020  51 8A C3 AA B8 FF FF AB 8B CA 33 C0 F3 AA 59 E2
100030  EF B9 1A 00 F3 A4 B8 20 00 F7 26 51 00 2D 1A 00
```

If a VDISK driver is already resident, the new program must inspect the fields within the driver itself and the boot block at the 1-megabyte boundary to determine the starting address of available extended memory, using the higher of the two if they are inconsistent. It must then decide how much memory to reserve for itself, and update the two fields just mentioned to reflect that amount of allocated extended memory.

If no VDISK driver is present in the system, the program can take the easy way out and hook the `Int 15H` vector to use the `Int 15H` method of memory management. Alternatively, it can pretend that it is a VDISK, pointing the `Int 19H` vector to something that appears to be a VDISK driver header, and creating a phony boot block at 1 megabyte. In either case, the program must also install its own Control-C (`Int 23H`) and critical-error (`Int 24H`) handlers so that it cannot be terminated unexpectedly.

Regardless of the allocation method used, the program must be careful to exit gracefully, so that it removes all evidence of its presence, and any extended memory it used is not orphaned. If the `Int 15H` vector was captured, the vector must be restored to point to the previous owner; if the VDISK indicators were modified, they must be returned to their proper state to "release" the memory. This can be quite tricky if another driver or TSR has allocated some extended memory to itself after the application program in question.

The eXtended Memory Specification (XMS)

The VDISK and Int 15H management methods described have serious weaknesses. First, neither method is immune to non-cooperating applications that simply switch into protected mode, find the size of extended memory by reading and writing it directly, and then use it all without regard to other, previously loaded programs. Second, neither management technique is dynamic; both allocate memory in a first-in-last-out manner. If a program terminates out of order, that program's extended memory is not available for use by other programs until all the extended memory that was allocated afterward by other applications is also released. Finally, MS-DOS does not participate in the expanded memory management, so it cannot "clean up" a program's expanded memory resources if the program terminates unexpectedly.

During 1988 (four years after the introduction of the PC/AT), two long overdue proposals for a more sophisticated, cooperative use of extended memory under DOS appeared. One of the proposals, the Virtual Control Program Interface, is applicable only to 80386/486-based systems, and is discussed in more detail in Chapter 8. The other is the eXtended Memory Specification (XMS), which was a collaborative effort of Microsoft, Intel, AST Research, and Lotus Corp.

The XMS defines a software interface for 80286-, 80386-, and 80486-based PCs that allows real-mode application programs to use extended memory, as well as certain areas of conventional memory not ordinarily managed by MS-DOS, in a cooperative and hardware-independent manner. The XMS defines functions calls that allocate, resize, and release memory blocks of three basic types:

- upper memory blocks (UMBs) at addresses between 640K and 1024K (1MB)
- the so-called "high memory area" (HMA) at addresses between 1024K and 1088K (more about this later)
- extended memory blocks (EMBs) from addresses above 1088K.

The XMS also provides hardware-independent control over the CPU's address line A20, which must be enabled to read or write extended memory. A summary of the XMS functions can be found in Table 3-2, and a complete description of the XMS programming interface can be found in Table 3-5 at the end of this chapter.

Table 3-2: Summary of functions defined by the Microsoft/Intel/Lotus/AST eXtended Memory Specification (XMS).

Function	Description
Driver information	
00H	Get XMS version
High memory area management	
01H	Allocate high memory area
02H	Free high memory area
A20 line management	
03H	Global enable A20 line
04H	Global disable A20 line
05H	Local enable A20 line
06H	Local disable A20 line
07H	Query A20 line state
Extended memory block (EMB) management	
08H	Query free extended memory
09H	Allocate extended memory block
0AH	Free extended memory block
0BH	Move extended memory block
0CH	Lock extended memory block
0DH	Unlock extended memory block
0EH	Get handle information
0FH	Resize extended memory block
Upper memory block (UMB) management	
10H	Allocate upper memory block
11H	Free upper memory block

Using XMS Support

An installable device driver that implements the XMS is called an eXtended Memory Manager (XMM). The prototype XMM provided by Microsoft, named HIMEM.SYS, is installed by adding a DEVICE= line to the CONFIG.SYS file, and rebooting the system. HIMEM.SYS accepts two optional switches on the DEVICE= line that loads the driver:

- /HMAMIN=*n* specifies the minimum number of kilobytes in the high memory area (HMA) that a program may use (0-63, default = 0).
- /NUMHANDLES=*n* sets the maximum number of XMS handles that may be active at any one time (0-128, default = 32).

A program can determine whether any XMM is available by setting AX to 4300H and executing an Int 2FH. If the driver is present, the value 80H is returned in AL; if the driver is absent, AL is returned unchanged or with some other value. For example:

```
        mov  ax,4300h       ; 4300H = get install status
        int  2fh            ; call driver
        cmp  al.80h         ; status = installed?
        je   present        ; yes, driver is present
        jmp  absent         ; no, driver is absent
```

This differs from the convention used by the MS-DOS extensions PRINT, SHARE, ASSIGN, and APPEND, which are also accessed via Int 2FH (using other values in AH, of course). These return AL = FFH if they are already installed.

After a program has established that the XMS driver is available, it obtains the entry point of the driver by executing Int 2FH with AX = 4310H. The entry point is returned in registers ES:BX and must be saved in a variable:

```
xmsaddr dd   ?              ; receives entry point
        .
        .
        .
                            ; get address of XMS
                            ; driver entry point...
        mov  ax,4310h       ; func. 43H subf. 10H
        int  2fh            ; invoke driver
        mov  word ptr xmsaddr,bx
                            ; save far pointer
        mov  word ptr xmsaddr+2,es
                            ; to entry point...
```

Once the entry point is in hand, the program enters the driver by a far call, without further resort to Int 2FH. A particular XMS function is selected by the value in AH; other parameters are passed in registers. At least 256 bytes of stack space should be available when your program requests an XMS function. The general form of the call is:

```
xmsaddr   dd   ?               ; receives entry point
          .
          .
          .
                               ; request XMS function...
          mov  ah,function     ; AH = function number
          .                    ; load other registers with
          .                    ; function-specific values
          call [xmsaddr]       ; indirect far call to driver
```

Most XMS functions return a status in register AX: `0001H` if the function succeeded, or `0000H` if the function failed. In the latter case, an error code is also returned in register BL with the high bit set (see Table 3-6 at the end of the chapter). Other results are also passed back in registers.

A typical program's use of an XMM to manage extended memory is shown in the following list:

1. Establish presence of the XMM using `Int 2FH`. If the XMM is not present, the program must determine whether it can continue without extended memory, or attempt to allocate extended memory using the VDISK or `Int 15H` methods described earlier in this chapter.
2. Allocate one or more extended memory blocks with XMS Function `09H`. The XMM returns a handle for each allocated block. (An extended memory block handle, like a file handle, is an arbitrary number, and its value has no direct relationship to the memory block's location).
3. Copy data between conventional memory and extended memory using XMS Function `0BH`. This function requires a parameter block that specifies the source and destination addresses in terms of a handle and a 32-bit offset. If a handle is nonzero, it refers to an allocated extended memory block, and the offset is from the base of that block. If the handle is zero, then conventional memory is being addressed, and the 32-bit offset position contains a far pointer in standard Intel format.
4. Before terminating, the program releases its extended memory block handle(s) with XMS Function `0AH`, so that the corresponding memory can be reused by other programs.

A code skeleton for the complete process of detecting the XMS driver, obtaining its entry point, allocating extended memory, and releasing extended memory is listed below:

```
reqsize equ     64                      ; KB extended memory to allocate
        .
        .
        .

xmm     dd      0                       ; far pointer to XMM entry point
total   dw      0                       ; total KB ext. memory avail.
largest dw      0                       ; size of largest block in KB
handle  dw      0                       ; extended memory block handle

movpars equ     $                       ; XMS Function 0BH param block
movelen dd      0                       ; length to move in bytes
shandle dw      0                       ; source handle
soffset dd      0                       ; source offset or far pointer
dhandle dw      0                       ; destination handle
doffset dd      0                       ; dest. offset or far pointer

mybuf   db      256 dup (?)             ; contains data to be moved to
                                        ; extended memory block
bufsize equ     $-mybuf                 ; length of data to be moved

        .
        .
        .
        mov     ax,4300h                ; check if XMM present
        int     2fh                     ; using multiplex interrupt
        cmp     al,80h                  ; status = installed?
        jne     error                   ; no, proceed

        mov     ax,4310h                ; XMM available, request
        int     2fh                     ; entry point via multiplex
        mov     word ptr xmm,bx         ; interrupt and save it
        mov     word ptr xmm+2,es

        mov     ah,8                    ; get available extended memory
        call    xmm                     ; transfer to XMM
        mov     total,dx                ; save total KB available
        mov     largest,ax              ; save largest free block
        cmp     dx,reqsize              ; enough memory?
        jb      error                   ; insufficient memory, jump

        mov     ah,9                    ; function 9 = allocate block
        mov     dx,reqsize              ; DX = block size desired in KB
        call    xmm                     ; transfer to XMM driver
        or      ax,ax                   ; allocation successful?
        jz      error                   ; jump if allocation failed
        mov     handle,dx               ; save extended memory handle
```

```
                                      ; set up param block for move
                                      ; from 'mybuf' to ext. memory
        mov     shandle,0             ; zero out source handle
        mov     word ptr soffset,offset mybuf   ; source address in
        mov     word ptr soffset+2,seg mybuf    ; conventional memory
        mov     ax,handle             ; destination handle for
        mov     dhandle,ax            ; extended memory block
        mov     word ptr doffset,0    ; destination 32-bit offset
        mov     word ptr doffset+2,0  ; within extended memory block
        mov     word ptr movelen,bufsize; length of data to move
        mov     word ptr movelen+2,0

        mov     ah,0bh                ; function 0BH = move data
        mov     si,offset movpars     ; DS:SI = param block address
        call    xmm                   ; transfer to XMM driver
        or      ax,ax                 ; any error?
        jz      error                 ; jump if error occurred

        mov     ah,0ah                ; function 0AH = release block
        mov     dx,handle             ; DX = extended memory handle
        call    xmm                   ; transfer to XMM driver
        or      ax,ax                 ; any error?
        jz      error                 ; jump if error occurred
```

As an alternative to using XMS Function 0BH, the program can "lock" its extended memory block with XMS Function 0CH, obtaining the current physical base address of the block. The program can then use Int 15H Function 87H to move data in or out of extended memory. When the program is finished with an access to extended memory, it unlocks its block again with XMS Function 0DH, allowing the XMM to move the allocated block around in order to coalesce free blocks and satisfy other allocation requests.

The High Memory Area

If you follow the trade press, you may remember a certain amount of publicity and claims of increased performance surrounding the release of Microsoft Windows version 2.1, which was concurrently—by a strange coincidence—renamed Windows/286. In its press releases, Microsoft stated that it had "found" an extra 64K of memory to put Windows kernel code in, and this allowed Windows to run much faster because it drastically reduced the amount of segment swapping. This mysterious 64K of memory, which Microsoft dubbed the "high memory area" (HMA), is actually the first 64K of extended memory, less 16 bytes. But how can it be possible for Windows/286, which is a real-mode program, to execute code out of extended memory?

The answer is clever, yet extremely simple. Recall the scheme by which physical addresses are generated in real mode: the contents of a segment register are shifted left four bits and added to a 16-bit offset. On an 8086/88 machine, if the result overflows the 20-bit addresses supported by the CPU, the address simply wraps; i.e., the upper bits are discarded. For example, an 8086/88 will interpret the address FFFF:FFFFH as 0000:FFEFH. Of course, 80286- and 80386/486-based PCs can support larger physical addresses (24 bits and 32 bits respectively), but this is ordinarily not apparent when MS-DOS is running, because these machines have special hardware to disable the most significant address lines in real mode, making them behave more like a classic 8086/88-based PC.

Now imagine the consequences if your program is running on an 80286-based PC and you enable the A20 line to allow the generation of 21-bit physical addresses, and then place the value FFFFH in one of the segment registers. When FFFFH is shifted left four bits and added to a 16-bit offset, the result is in the range FFFF0H–10FFEFH. In other words, enabling the A20 line allows the first 65,520 bytes of extended memory to be addressed *without leaving real mode.*

The XMS specification bears on the discovery of the HMA in two ways. First, it provides a hardware-independent method of enabling or disabling the A20 line. This eliminates the need for programs to write directly to the ports that control the A20 line (possibly interfering with each other, especially in the case of interrupt handlers), and ensures that the toggling of the A20 line is always done in the most efficient way. Second, it arbitrates the use of the high memory area between competing programs.

The management of the high memory area is not very complex, since the HMA is so small, and it is always allocated as a unit. A device driver or TSR program that uses the HMA should store as much of its code there as possible, since the remainder will simply be lost for use by other programs. If the driver or TSR cannot exploit nearly all of the HMA, it should leave it available for use by subsequently loaded programs. The user can enforce such good behavior with the /HMAMIN switch, which causes allocation requests for the HMA to fail if they are smaller than the specified value.

Device drivers and TSRs must not leave the A20 line permanently turned on. Although it might seem difficult to believe, some applications rely on the wrapping of memory addresses at the 1-megabyte boundary, and will overwrite the HMA instead if the A20 line is left enabled. Similarly, interrupt vectors must not point directly into the HMA, since the A20 line will not necessarily be enabled at the time that the interrupt is received, so the code that comprises the interrupt handler might not be visible. If the HMA is still available when a normal application runs, the application is free to use as much or as little of the HMA as it wishes, with the following restrictions:

- Far pointers to data located in the HMA cannot be passed to MS-DOS since MS-DOS normalizes pointers in a manner that invalidates HMA addresses.
- Disk I/O directly into the HMA by any method is not recommended. The behavior of some clone disk controllers—when handed addresses that fall within the HMA—may vary.

An application that finds the HMA available and allocates it must also be sure to release it before terminating. Otherwise, the HMA will be unavailable for use by any other program until the system is restarted.

LOADALL: The Back Door to Extended Memory

There are two methods by which programs can obtain access to data in extended memory while the CPU is in real mode. The first of these methods, which relies on placing the special value `FFFFH` in a segment register along with manipulation of the bus's A20 address line, has already been described in the section on the *High Memory Area*. Unfortunately, this technique only provides access to the first 65,520 bytes of extended memory. The second method, which employs the 80286's undocumented LOADALL instruction, can be used to reach *any* location in extended memory.

To understand how LOADALL can provide this magical capability, we must first recall how the Intel CPUs generate physical memory addresses. In real mode, the contents of a segment register is shifted left by four bits (i.e., multiplied by 16) and added to a 16-bit offset to form a 20-bit physical address. In protected mode, an additional layer of address indirection is added. The upper 13 bits of the segment register are used as an index into a *descriptor table*—a special data structure that is manipulated by the operating system and interpreted by the hardware—and a 24-bit physical memory address is generated by combining a base address from a descriptor with a 16-bit offset.

Fortunately, while the explanation in the preceding paragraph is correct and useful in the abstract, it is not a *complete* description of how the CPU produces physical memory addresses. Imagine the penalty in CPU cycles and execution time if the CPU actually had to perform a 4-bit shift on the contents of a segment register each time a program referenced memory in real mode! Worse yet, try to envision the cost in CPU cycles and bus traffic if the CPU had to fetch a 24-bit physical address from a descriptor each time a program accessed memory in protected mode! By looking in the Intel manuals, however, we can see that the cost of a memory reference in protected mode is usually the same as in real mode (unless a segment register is also being loaded, as in the case of "far" JMPs and CALLs), which is a clue that something else must be going on.

This something else turns out to involve the existence of a set of *shadow registers* on the CPU chip called *descriptor caches*—one for each segment register. Whenever a segment register is loaded with a POP or MOV instruction, the CPU calculates (in real mode) or fetches (in protected mode) the true physical base address and length of the designated memory segment, and caches these values in the associated shadow register. Subsequently, each time the segment register is referenced by an instruction that accesses memory, the CPU simply adds the base address from the descriptor cache to the offset specified in the instruction to quickly form the final physical memory address.

The essential action of the LOADALL instruction is to initialize the contents of every CPU register and flag including the descriptor caches we have just been discussing from a 102-byte table stored in a specific format at physical memory address 00800H (see Tables 3-3 and 3-4). It seems that the original intent of the LOADALL instruction was only to aid in CPU testing, which is why it was never included in any Intel manuals. But since LOADALL allows arbitrary physical base addresses to be forced into the shadow registers, it can also be exploited by a real-mode application program to read or write memory locations that would not otherwise be addressable.

The LOADALL instruction is not supported by the Microsoft Macro Assembler, but you can include its op-code (0FH 05H) in your programs with DB statements. LOADALL must be used with great caution though. If an interrupt occurs after you execute LOADALL, but before you complete the access to extended memory, the interrupt handler may load the segment register and thus change the contents of the associated descriptor cache, and your extended memory read or write will go astray. Therefore, interrupts must be blocked throughout the execution of code that relies on LOADALL. Furthermore, the 102 bytes starting at address 00800H lie within memory controlled by MS-DOS, so you must carefully save and restore this area.

Assuming LOADALL is used cautiously, can it be used safely? That is, can we expect a program containing the LOADALL instruction to run correctly and reliably on a range of DOS versions, PC clone brands, and hardware configurations? The answer, at least on 80286-based PCs, seems to be a qualified *yes*. Microsoft uses LOADALL in the RAMDRIVE.SYS virtual disk driver supplied with Windows and the OEM versions of MS-DOS, and also uses it in the DOS compatibility environment of OS/2, so we can predict (given Microsoft's close relationship with Intel) that LOADALL isn't likely to vanish from future steppings of Intel's 80286 chips. For the same reason, the 80286 CPUs from second sources such as AMD and Harris will be obligated to support LOADALL indefinitely.

On 80386- or 80486-based PCs, the answer is not so clear-cut. The Intel 80386 and 80486 CPUs do not have a LOADALL instruction, so execution of LOADALL triggers an *invalid op-code* exception. In order for programs containing LOADALL to run properly, the ROM BIOS must field the exception, examine the instruction that caused the interrupt, and emulate the action of LOADALL if necessary. High-quality ROM BIOSes (such as those found on Compaq 80386 and 80486 machines) can be relied on in this area but other companies' ROM BIOSes are not as predictable, which is one of several reasons why OS/2 doesn't run on many PC/AT clones.

Table 3-3: The data structure used by the undocumented 80286 LOADALL instruction. This structure must always be located at physical memory address 00800H, and is used to initialize all CPU registers and flags.

Memory Access	CPU Register
0800-0805H	none
0806-0807H	MSW (Machine Status Word)
0808-0815H	none
0816-0817H	TR Register (Task Register)
0818-0819H	CPU Flags Word
081A-081BH	IP Register (Instruction Pointer)
081C-081DH	LDTR Register (Local Descriptor Table Register)
081E-081FH	DS Register
0820-0821H	SS Register
0822-0823H	CS Register
0824-0825H	ES Register
0826-0827H	DI Register
0828-0829H	SI Register
082A-082BH	BP Register
082C-082DH	SP Register
082E-082FH	BX Register
0830-0831H	DX Register
0832-0833H	CX Register

Memory Access	CPU Register
0834-0835H	AX Register
0836-083BH	ES Descriptor Cache
083C-0841H	CS Descriptor Cache
0842-0847H	SS Descriptor Cache
0848-084DH	DS Descriptor Cache
084E-0853H	GDTR (Global Descriptor Table Register) Cache
0854-0859H	LDTR (Local Descriptor Table Register) Cache
085A-085FH	IDTR (Interrupt Descriptor Table Register) Cache
0860-0865H	TSS (Task State Segment) Descriptor Cache

*See Table 3-4 for the format of the fields for the descriptor cache, GDTR cache, LDTR cache, and IDTR cache.

Table 3-4: The format of the 6-byte fields in the LOADALL data structure which are used to load the CS, DS, ES, and SS descriptor caches, GDTR cache, LDTR cache, and IDTR cache.

Offset	Contents
0-2	24-bit segment base address, with least significant byte at lowest address, and most significant byte at highest address
3	Access rights byte for CS, DS, ES, and SS descriptor caches; 0 for GDTR, LDTR, and IDTR caches
4-5	16-bit segment size

Table 3-5: The XMS Programming Interface

Function	Parameters	Results if Successful	Results if Unsuccessful

XMS Function 00H
Get XMS Version
	AH = 00H	AX = XMS version
		BX = XMM (driver) version
		DX = HMA indicator
		0000H if no HMA
		0001H if HMA exists
			AX = 0000H
			BL = error code

Note: Version numbers are binary coded decimal (BCD). The value returned in DX is not affected by any previous allocation of the HMA by another program.			

XMS Function 01H
Allocate High Memory Area (HMA)
	AH = 01H
	DX = HMA bytes needed (driver or TSR) or 0FFFFH (application program)
		AX = 0001H	AX = 0000H
			BL = error code

Function	Parameters	Results if Successful	Results if Unsuccessful
Note: The maximum HMA allocation is 65,520 bytes. The base address of the HMA is 0FFFF:0010H. If an application fails to release the HMA before it terminates, the HMA becomes unavailable to other programs until the system is restarted.			

XMS Function 02H
Free High Memory Area (HMA)
	AH = 02H	AX = 0001H	AX = 0000H
			BL = error code

XMS Function 03H
Global Enable A20 Line
	AH = 03H	AX = 0001H	AX = 0000H
			BL = error code

Note: This function should only be used by programs that have successfully allocated the HMA. The A20 line should be disabled before the program releases control of the system.			

XMS Function 04H
Global Disable A20 Line
	AH = 04H	AX = 0001H	AX = 0000H
			BL = error code

Note: This function should only be used by programs that have successfully allocated the HMA.			

XMS Function 05H
Local Enable A20 Line
	AH = 05H	AX = 0001H	AX = 0000H
			BL = error code

Note: This function should be used by programs that do not own the HMA. The A20 line should be disabled before the program releases control of the system.			

XMS Function 06H
Local Disable A20 Line
	AH = 06H	AX = 0001H	AX = 0000H
			BL = error code

Note: This function should be used by programs that do not own the HMA.			

XMS Function 07H
Query A20 Address Line Status
	AH = 07H	*If A20 line is enabled*
		AX = 0001H
		If A20 line is disabled
		AX = 0000H
		BL = 00H
			AX = 0000H
			BL = error code

XMS Function 08H
Query Free Extended Memory
	AH = 08H	AX = largest free extended memory block (KB)
		DX = total free extended memory (KB)
			AX = 0000H
			BL = error code

Note: The size of the HMA is not included in the returned values, even if it is not in use.			

XMS Function 09H
Allocate Extended Memory Block (EMB)
	AH = 09H
	DX = requested block size (KB)
		AX = 0001H
		DX = EMB handle
			AX = 0000H
			BL = error code

Note: An EMB block length of zero is explicitly allowed.			

Function	Parameters	Results if Successful	Results if Unsuccessful

XMS Function 0AH
Free Extended Memory Block (EMB)
	AH = 0AH
	DX = EMB handle
		AX = 0001H	AX = 0000H
			BL = error code

Note: If an application fails to release its extended memory before it terminates, the memory becomes unavailable for use by other programs until the system is restarted.			

XMS Function 0BH
Move Extended Memory Block (EMB)
	AH = 0BH
	DS:SI = segment:offset of parameter block
		AX = 0001H	AX = 0000H
			BL = error code

Note: Parameter block format: *dword* length of block (bytes) *word* source EMB handle *dword* source offset *word* destination EMB handle *dword* destination offset. If source and/or destination handle is zero, the corresponding offset is assumed to be a normal far pointer. The EMB need not be locked. The state of the A20 line is preserved.			

XMS Function 0CH
Lock Extended Memory Block (EMB)
	AH = 0CH
	DX = EMB handle
		AX = 0001H
		DX:BX = 32-bit linear address of locked block
			AX = 0000H
			BL = error code

Note: This function is intended for use by programs which enable the A20 line and then access extended memory directly. Lock calls may be nested.			

XMS Function 0DH
Unlock Extended Memory Block (EMB)
	AH = 0DH
	DX = EMB handle
		AX = 0001H	AX = 0000H
			BL = error code

Note: After an EMB is unlocked, the 32-bit linear address returned by any previous lock call becomes invalid and should not be used.			

XMS Function 0EH
Get EMB Handle Information
	AH = 0EH
	DX = EMB handle
		AX = 0001H
		BH = lock count (0 if block not locked)
		BL = number of handles still available
		DX = block size (KB)
			AX = 0000H
			BL = error code

XMS Function 0FH
Resize Extended Memory Block (EMB)
	AH = 0FH
	BX = new block size (KB)
	DX = EMB handle
		AX = 0001H	AX = 0000H
			BL = error code

Note: Blocks may not be resized while they are locked.			

Function	Parameters	Results if Successful	Results if Unsuccessful

XMS Function 10H Allocate Upper Memory Block (UMB)	AH = 10H
	DX = requested block size (paragraphs)
		AX = 0001H
		BX = segment base of allocated block
		DX = actual block size (paragraphs)
			AX = 0000H
			BL = error code
			DX = size of largest available block (paragraphs)

Note: Upper memory blocks are always paragraph aligned. The A20 line need not be enabled to access an UMB.			

XMS Function 11H Free Upper Memory Block (UMB)	AH = 11H
	DX = segment base of block
		AX = 0001H	AX = 0000H
			BL = error code

Table 3-6: XMS error codes.

Value	Meaning
80H	Function not implemented
81H	VDISK device driver was detected
82H	A20 error occurred
8EH	General driver error
8FH	Unrecoverable driver error
90H	High memory area does not exist
91H	High memory area already in use
92H	DX is less than /HMAMIN= parameter
93H	High memory area not allocated
94H	A20 line still enabled
A0H	All extended memory is allocated
A1H	Extended memory handles exhausted
A2H	Invalid handle
A3H	Invalid source handle
A4H	Invalid source offset
A5H	Invalid destination handle
A6H	Invalid destination offset
A7H	Invalid length
A8H	Invalid overlap in move request
A9H	Parity error detected
AAH	Block is not locked
ABH	Block is locked
ACH	Lock count overflowed

Value	Meaning
ADH	Lock failed
B0H	Smaller UMB is available
B1H	No UMBs are available
B2H	Invalid UMB segment number

Chapter 4

80286-based Protected-Mode DOS Extenders

Andrew Schulman

Several software manufacturers sell products that allow programs written for MS-DOS to access up to 16 megabytes of memory, in contrast to the 640K limit of MS-DOS. Unlike EMS or XMS, the memory access these products provide is transparent, in that "normal" pointers can be used. Programs developed with these products continue to use MS-DOS, but run in the "protected mode" of the 80286 and higher Intel microprocessors. We refer to these products as 80286-based protected-mode DOS extenders.

Lotus 1-2-3 Release 3 is one example of a program that uses an 80286-based protected-mode DOS extender, Rational Systems' DOS/16M. Other products that employ DOS/16M include AutoCAD Release 10.0 (AutoDesk), the TOPS network (Sun/TOPS), Informix SQL and Informix 4GL (Informix), Glockenspiel C++ (ImageSoft), and Rational Systems' own Instant-C. DOS/16M is planned for inclusion in the next release of Ashton-Tate's dBASE IV. DOS extenders—once obscure boutique items—have moved into the mainstream, and even the forefront, of commercial PC software development.

By 80286-based, we mean software that requires *at least* an IBM PC/AT or compatible, and that runs in 16-bit protected mode. Programs such as the MS-DOS version of Lotus 1-2-3 Release 3 also run on PC-compatible computers with Intel 80386 or 80486 CPUs. Just as these 32-bit protected-mode processors can lower themselves by emulating the real-mode 8088, so too can they emulate the 16-bit protected mode native to the 80286.

Why develop for the 286?

The 80286 is on the way out. Even Intel, which designed the 286, is running ads that flatly state, "It just doesn't make sense to buy another 286-based personal computer," and that encourage you to "invest in the future, not in the past."

While Intel may have ulterior motives in downplaying the 286 (several other companies now produce 286 chips), the company's ads are right. You probably shouldn't buy a 286-based computer. So why produce 286-based software?

Because right now, if you are developing software to run under real-mode MS-DOS, you are producing 8088-based software. 286-based protected-mode software may not sound "cutting edge," but it is way ahead of where most PC software is today. Without some form of protected-mode operating environment, such as OS/2 or a DOS extender, even the fastest 80486 can only be used as a fast 8088. Without protected mode, the new machines are all dressed up with nowhere to go. Protected mode junks 8088 compatibility.

Why not go straight to 386-based protected mode? Doesn't developing for a 286-based DOS extender repeat the mistake that Microsoft and IBM apparently made when they developed OS/2 for the 286 instead of the 386?

It all depends on your application. If you can guarantee that your potential customers have 386 computers, then 32-bit protected mode is the way to go. Otherwise, the 16-bit protected mode of the 286 is a better base. While there are still only two million 386-based PCs currently in use worldwide, there are about *twelve million* 286-based PC/ATs and compatibles.

After all, why is there so little 80386-dependent software for all the 80386 hardware that the computer trade press is urging we buy? Because of *compatibility*. As long as there are XTs and ATs out there, software vendors are rightly hesitant to lock themselves solely into a 386 market.

By using a 286-based protected-mode DOS extender like DOS/16M, you cut yourself off from customers with XTs, but not from customers with ATs. Thus, 16-bit 286-based DOS extenders appear to be a good compromise between the desire for software to finally catch up with hardware, and the desire not to be locked in to the small (though rapidly growing) 386 market.

In many ways, a DOS extender combines the best of both worlds: continued access to MS-DOS (`Int 21H`) and BIOS services, but with the ability to develop multi-megabyte programs and use the native protection capabilities of the Intel microprocessors, which lie fallow in real mode.

Focusing on Rational Systems' DOS/16M, which provides up to 16 megabytes of memory while running under MS-DOS 3.x and higher, this chapter shows you how to reap the benefits of DOS extenders. First, it shows how a 286-based DOS extender works, and how to port programs from real mode to this protected-mode MS-DOS hybrid, with a minimum of changes. (Often, this barely merits being called a port, since in many cases the 286-based DOS extender version of your program can use the same .OBJ files as the real-mode version.) Finally, it shows how to eliminate protection violations and how to improve performance.

Since the primary benefit of a DOS extender is access to multi-megabytes of memory, very large programs have the most to gain from using a DOS extender. (Though programs with a small amount of code, but very large data requirements, clearly also benefit.) Such large programs are frequently written in C, so this chapter contains a number of sample programs in C.

On the other hand, when you port to protected mode, the few thorny areas tend to be confined to a small part of the program written in assembler, so we use assembler examples as well.

Rational Systems also makes a protected-mode integrated development environment, Instant-C, which runs under DOS/16M. We use Instant-C examples in an appendix to this chapter; its interactive style provides a convenient base for exploring protected mode, and its price is substantially less than that of DOS/16M. While not intended as a substitute for DOS/16M, it can run most of the sample code in this chapter.

This chapter also examines Eclipse Computer Solutions' OS/286, which provides many of the same capabilities as DOS/16M. Most of the programs in this chapter can also be compiled for OS/286. But there are important differences between DOS/16M and OS/286. One key difference is that DOS/16M is much easier to use than OS/286, while OS/286 is much cheaper than DOS/16M. In addition to applications like CadKey 3 Plus, OS/286 is incorporated in a number of programming languages, such as Golden Common Lisp (Gold Hill), Lahey Fortran 77L-EM/16, and the Lattice 80286 C Development System.

Most of the changes required when porting your program from real mode to a protected-mode DOS extender are also required when porting to OS/2. In fact, most of the principles involved in programming for a 286-based DOS extender apply to *any* 16-bit protected-mode environment, including OS/2.

While this chapter emphasizes porting code from real-mode MS-DOS to a protected-mode DOS extender, a different scenario involves porting a mainframe application, which, without a DOS extender or other protected-mode environment, would never be able to run on a PC/AT. Such ports should be far simpler than those described here.

Protected-Mode MS-DOS

To demonstrate how programming for a 286-based DOS extender differs from "normal" DOS programming, it is useful to construct a program that manipulates a large amount of data. One of the touted benefits of DOS extenders is that they break the 640K barrier, so we need to see how difficult it is to get at this extra memory, and what special steps are involved.

The following C program builds a linked list as large as available memory. It allocates nodes and adds them to the linked list until the C memory-allocation function `malloc()` returns NULL, indicating that memory is exhausted. The program prints out the number of nodes in the list and how many bytes of memory it has allocated. It then walks back through the list, using the `free()` function to deallocate the nodes:

```
/*
LIST.C

Microsoft C 5.1 real mode:
    cl -AL -Ox -W3 list.c

DOS/16M protected mode:
    ; use same OBJ file; just relink and postprocess
    link /noe/map \16m\preload \16m\crt0_16m \16m\pml list,list,list;
    \16m\makepm list
    \16m\splice list.exe list.exp \16m\loader.exe

OS/286 protected mode:
    ; use same OBJ file; just relink and postprocess
    link /noe/map list,list,list,\os286\llibce.lib;
    \os286\express list
    \os286\bind -o list.exe -l \os286\tinyup.exe \
        -k \os286\os286.exe -i list.exp

Turbo C real mode:
    tcc -ml list

DOS/16M Turbo C version:
    ; use same OBJ file; just relink and postprocess
    tlink /m \16m\tc\preload \16m\tc\c0l list \16m\tc\pml \
        \16m\tc\setargv \16m\tc\mem_16m,list,list,\tc\lib\cl;
    \16m\makepm -stack 8192 list
    \16m\splice list.exe list.exp \16m\loader.exe

to run:
    LIST [node size]

output on 2meg Compaq:
    real mode:  527k
    DOS/16M:    1692k
    OS/286:     1372k
*/

#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <time.h>

typedef struct node {
    unsigned long num;
    void *data;
    struct node *next;
    } NODE;

main(int argc, char *argv[])
{
    NODE *p, *q;
    time_t t1, t2;
    unsigned long nodes = 0;
    unsigned nodesize = (argc > 1) ? atoi(argv[1]) : 512;

    time(&t1);

    /* allocate linked list that consumes all available memory */
    for (q = NULL; ; q->next = p)
    {
        p = q;

        if ((q = malloc(sizeof(NODE))) == NULL)
            break;
        if ((q->data = malloc(nodesize)) == NULL)
        {
            free(q);
            break;
        }
        q->num = nodes++;
        if ((nodes % 1000) == 0)
```

```
            printf("%lu nodes: %lu seconds\n", nodes, time(&t2) - t1);
    }

    printf("%lu nodes: %lu seconds\n", nodes, time(&t2) - t1);
    printf("Allocated %uK\n", (nodes * (sizeof(NODE)+nodesize)) >> 10);

    /* in reverse order, deallocate the nodes */
    for ( ; p != NULL; p = q)
    {
        q = p->next;
        if (p->num != --nodes)
            printf("list corrupt: nodes=%lu num=%lu\n", nodes, p->num);
        free(p->data);
        free(p);
    }

    /* zero nodes remaining indicates success */
    return nodes;
}
```

When this program is compiled for real-mode MS-DOS, using large model (which manipulates far pointers), and run on an IBM PC/AT with two megabytes of memory, the program allocates about 550K, oblivious to the presence of more memory in the machine. This real-mode version can be compiled with Microsoft C 5.1, for example, using the following command-line:

```
cl -AL -Ox -W3 list.c
```

The `cl` driver program first runs the C compiler to produce LIST.OBJ, and then runs the Microsoft linker to produce LIST.EXE.

Surprisingly, to make a protected-mode version of the same program, we can use the exact same LIST.OBJ that the compiler produced for real mode. These are the commands to produce a DOS/16M version of LIST.EXE, using the object module LIST.OBJ:

```
link /noe/map \16m\preload \16m\crt0_16m \16m\pml list,list,list;
\16m\makepm list
\16m\splice list.exe list.exp \16m\loader.exe
```

Since this protected-mode LIST.EXE uses the same LIST.OBJ as the real-mode LIST.EXE, it is a little difficult to believe that it behaves any differently. Nevertheless, it does. On the same 286 machine with 2 megabytes of memory, the DOS/16M version of LIST.EXE allocated 1692K, more than three times as much as in real mode. On a machine with more memory, the protected-mode program allocates an even longer (up to 16 megabytes) linked list, whereas in real mode it is always stuck at around 550K.

With the same .OBJ as in real mode, it's obvious that no additional code has been written to access extended memory. Contrast this to DOS extensions such as EMS or XMS, which also provide access to more memory, but which do so *indirectly*. As explained in Chapters 2 and 3, in order to allocate expanded memory with EMS or extended memory with XMS, you need to make separate calls to these memory managers. In real mode, for example, the C memory-allocation function `malloc()` cannot allocate out of EMS or XMS memory.

And allocation is just the beginning. When *using* memory in a high-level language, you would like to use a simple pointer dereference. But EMS requires that you map the logical page to a physical page, and XMS requires that you move from extended to conventional memory, before you can access memory allocated using these specifications.

In contrast, LIST.C freely uses, say, `p->num`, without knowing whether it is in extended or conventional memory. In fact, with a DOS extender, this distinction nearly disappears: it's all just memory.

This is big memory, but not *virtual* memory (VM). Here is one of several important differences between a DOS extender and a genuine operating system such as OS/2. Rational Systems does offer VM as an add-on to DOS/16M. But as built here, while LIST.EXE uses all available memory, it doesn't expand onto your hard disk, as it would under OS/2.

Note also that slightly different coding of LIST.C could have produced incompatibilities with protected mode. LIST.C uses the `time()` function to calculate how long memory allocations take in protected versus real mode. (It takes about the same amount of time per allocation.) Now if, instead of calling `time()`, we peeked at low-memory BIOS location `46CH`, this could, depending on how we formed the pointer, generate a protection violation under a DOS extender. Later on, we will see why this is so, and will show how you can safely peek directly at absolute physical locations under a protected-mode DOS extender.

In the three lines used to produce the DOS/16M version of LIST.EXE, we first used the standard Microsoft linker to relink LIST.OBJ along with some additional .OBJ files supplied with DOS/16M. The resulting file, LIST.EXE, is then run through a postprocessor, MAKEPM.EXE, that (as its name implies) makes a protected-mode (PM) executable from a real-mode executable. The output from this process is LIST.EXP. The extension .EXP designates a protected-mode executable.

Now, how do we run one of these .EXP files?

On the one hand, it would seem that we can't run it simply by typing its name on the DOS command line, since this is a protected-mode program, whereas MS-DOS is a real-mode operating system. On the other hand, DOS/16M is *not* an environment like DESQview or Windows: the user of a program built with a DOS extender does not go out and buy a special run-time shell.

Instead, from DOS you run a small (38K) program supplied with DOS/16M, called LOADER.EXE, which manages the interface between MS-DOS and protected mode.

But wait a minute—you don't want the users of your program to have to run some other program first just in order to run yours. To solve this problem, all DOS extenders provide a program that binds the loader/kernel together with your program, so that users *can* run your program from the DOS command line, simply by typing its name. In DOS/16M, this binder is SPLICE.EXE. In our example, SPLICE binds LIST.EXP together with LOADER.EXE to form LIST.EXE.

The file sizes involved are fairly reasonable. Whereas the real-mode LIST.EXE produced by Microsoft C 5.1 is 12K, protected-mode LIST.EXP is 13K; when SPLICE merges this with the 38K LOADER.EXE, the resulting protected-mode LIST.EXE is 51K: not bad for a program that has the convenience of MS-DOS without its restrictions.

How does it work?

It may seem rather odd to be reading about protected mode in a book on MS-DOS. After all, this is the sort of discussion usually found in books on OS/2 or on the Intel 286/386 architecture.

The strange thing about DOS extenders is that they provide a way to take advantage of the large address space and protection mechanism discussed in Chapter 1, all the while employing the services of real-mode MS-DOS.

This interface between MS-DOS and protected mode can be made to sound magical. In fact, the mechanism is rather simple.

Let's look behind the scenes and see how this works. In the remainder of this section, it is important to emphasize that we are talking about what a DOS extender like DOS/16M does, *not* what you have to do. All you have to do is more-or-less blindly follow the three steps—relink, postprocess, splice—and the DOS extender takes care of the rest. "The rest" is what we will now proceed to relate.

A protected-mode loader like DOS/16M's LOADER.EXE starts off running in real mode under MS-DOS. The loader constructs a global descriptor table (GDT), local descriptor table (LDT), and interrupt descriptor table (IDT) for your program, switches the machine into protected mode, and then spawns your program. By setting up the IDT, the loader sets up interrupt handlers for MS-DOS (`Int 21H`) and BIOS services (`Int 10H`, `Int 16H`, etc.). Whenever your program makes a DOS or BIOS request, the DOS/16M kernel's handler catches it and acts either as a front end to, or as a replacement for, the corresponding interrupt service routine in real mode. When your program exits, DOS/16M puts the computer back into real mode and exits back to DOS.

A DOS extender is not an operating system. In contrast to OS/2, a DOS extender exists simply to provide the minimal facilities for running your DOS program in protected mode. Thus, the descriptor tables need serve only your program; running only one task simplifies things considerably (though this is one of the important differences between DOS/16M and OS/286, which we will discuss later).

Protected-mode descriptor tables can be built in real mode (in fact, the GDT *must* be built in real mode, which is why 286 and 386 machines boot up in real mode, even though their "native" mode is protected); the GDT register (GDTR) and IDT register (IDTR) are each loaded with six bytes containing the size and physical base address of the corresponding table, and the LDT register (LDTR) is loaded with a selector to the LDT. For example:

```
lgdt fword ptr gdt_desc    ; load GDT
lidt fword ptr idt_desc    ; load IDT
lldt ax                    ; load LDT
```

Now that there is a GDT, the computer can be put into protected mode by setting the bottom bit of the machine status word (MSW), using the following instructions (the `jmp` clears the 286 instruction pipeline):

```
smsw ax     ; store MSW
or al, 1    ; set protected-mode bit
lmsw ax     ; load MSW
jmp $+2     ; clear pipeline
```

The machine is now in protected mode, and the DOS extender can spawn your protected-mode program.

The IDT is crucial to the operation of a DOS extender. It serves the same purpose in protected mode as the low-memory interrupt vector table in real mode, except that the IDT is composed of eight-byte *gates*, descriptors used to control access to code segments. In building an IDT for your program, a DOS extender most importantly *includes a descriptor for* `Int 21H`. This descriptor points not to the real-mode entry point for Int 21H (which could be MS-DOS itself or some other program that has hooked Int 21H, such as Sidekick), but to the DOS extender's protected-mode Int 21H handler. Thus, a DOS extender hooks Int 21H by setting up an interrupt gate in a protected-mode IDT, rather than by calling Int 21H AH=25H (Set Vector) or directly poking the low-memory interrupt vector table. The original Int 21H vector in the real-mode interrupt vector table is left alone.

Whenever your protected-mode program issues an Int 21H (to open a file, say, or to allocate memory), the DOS extender catches it, checks the function request in the AH register, and acts either as a replacement for, or as a front end to, real-mode MS-DOS. The DOS extender can service the request itself, or it can modify it, switch the machine back into real mode, resignal the Int 21H (in DOS/16M this is called a *passdown* interrupt), modify the return value, and switch the machine back into protected mode. This happens "inside" the Int instruction, without your program's knowledge.

This procedure works much like normal interrupt chaining, in which a program that has hooked an interrupt also passes it along to the previous owner. The difference here is that a CPU mode switch takes place before passing the interrupt down the chain.

You might have doubts about switching the machine from protected down to real mode. Switching from real to protected mode isn't a problem, and on a 386 switching from protected back to real mode is a simple matter as well, but the 286 is like a cat that knows how to climb up a tree but doesn't know how to get back down: it doesn't allow for switching back into real mode. The only way to make the transition is by resetting the chip. DOS extenders on the 286 use the same weird, but effective, "triple fault" technique for switching into real mode that Microsoft uses in the OS/2 compatibility box. One way to force a triple fault is to issue an Int 3H after setting the IDT limit to zero.

Can this really take place quickly enough? As measured by DOS/16M's PMINFO utility, while a 16 MHz Compaq 386 can switch back and forth between real and protected mode over 7,000 times per second, an 8 MHz IBM PC/AT can only switch about 1,200 times per second. But this is more than adequate for most applications. A program that needed to telecommunicate at 9600 baud on an IBM AT could avoid the costly gear shift, using several techniques documented in the DOS/16M and OS/286 manuals.

Getting back to our protected-mode program, when it finally exits back to DOS with Int 21H AH=4CH, it is actually exiting back to the DOS extender. The

DOS extender cleans up its descriptor tables, puts the machine back into real mode, and exits back to the "real" MS-DOS.

So that is how protected-mode programs can be run from the DOS command line and use DOS services. But the `Int 21H` program interface is not sufficient. The same mechanism used with `Int 21H` is also used for BIOS services such as the keyboard (`Int 16H`) and video (`Int 10H`). Programmers can use the same mechanism to communicate with other real-mode services (for example, the mouse, the DESQview API, or NetBIOS), or to call functions in a real-mode library (for example, a graphics library that requires conversion to run in protected mode, but for which you lack source code).

Even this is insufficient. Only the rarest well-behaved DOS program uses just DOS and BIOS calls. The IN and OUT instructions don't present a problem, since, for example, port `20H` is as valid in protected as in real mode. But most applications peek and poke various well-known memory locations. So a second technique allows protected-mode programs to use many protected-mode selectors as though they were well-known real-mode segment addresses. For instance, protected-mode selector `B800H` can be made to correspond to real-mode physical address `B8000H`. Similar to bimodal pointers used in OS/2 device drivers, these are referred to as transparent addresses by DOS/16M.

Now, isn't this all a kludge? In one way it certainly is, since MS-DOS was never intended to be called from protected mode. On the other hand, numerous programs piggyback `Int 21H`. By providing system services via interrupts, and by allowing complete flexibility in getting and setting the interrupt vectors, MS-DOS provides a powerful mechanism for patching and extending itself. DOS extenders are merely using this aspect of the PC architecture.

DOS extenders treat MS-DOS as a place to plug in an installable memory-management system. This is possible because of the interrupt-based architecture of PC system services. The existence of the MS-DOS Set Vector function means that such services, including `Int 21H` itself, are assumed to be replaceable, patchable, and chainable. (Remember, though, that the DOS extender plugs in its replacements by setting up a protected-mode IDT.)

To examine the interface a DOS extender provides between protected mode and MS-DOS, and to show that the DOS extender does its work, not through libraries or startup code, but at a much lower level, we can run a small assembler program under the DOS/16M debugger, Instant-D.

This program, FILEREAD.ASM, merely reads its own source code from disk and displays it on `stdout`; it can be assembled with the Microsoft assembler

(MASM) or Turbo assembler (TASM), linked with a DOS/16M module (PRELOAD.OBJ) that contains placeholders for the GDT, IDT, and other segments, and then run through the MAKEPM utility. The resulting program, FILEREAD.EXP, is 368 bytes. The source code can also be used for a real-mode version or for an Eclipse OS/286 version:

```
;       fileread.asm
;
;       real-mode version:
;       masm -Zi fileread;
;       link /co fileread;
;       cv fileread
;
;       DOS/16M version:
;       masm -Zi fileread;
;       link /co \16m\asm\preload fileread,fileread;
;       \16m\makepm fileread
;       \16m\d fileread
;
;       OS/286 version (within CP environment):
;       \os286\cp
;       masm -Zi fileread;
;       link /co/map fileread,fileread,fileread;
;       \os286\express -nsg -wb fileread
;       \os286\symtab fileread
;       load fileread

        dosseg
        .model small

        .stack

        .data
fname   db  "fileread.asm", 0

        .code
start:  mov dx, dgroup
        mov ds, dx

;       to allocate memory under DOS, first shrink down image
;       not needed for DOS extender, but doesn't hurt
        mov ah, 4ah
        mov bx, 100h
        int 21h                 ; ignore any errors here

        mov ah, 48h             ; allocate buffer
        mov bx, 100h
        int 21h
        jc  error
        mov di, ax              ; di = buffer selector

        mov ah, 3dh             ; open file
        mov al, 0
        mov dx, offset fname
        int 21h
        jc  error
        mov si, ax              ; si = file handle

        mov bx, si
        mov ah, 3fh             ; read file into buffer
        mov cx, 1000h
        push ds
        mov ds, di
        mov dx, 0
        int 21h
        pop ds
        jc  error

        mov cx, ax              ; ax = count of bytes read
        mov ah, 40h
        mov bx, 1               ; write buffer to stdout
        push ds
        mov ds, di
        mov dx, 0
        int 21h
        pop ds
        jc  error

        mov ah, 3eh             ; close file
        mov bx, si
        int 21h
        jc  error

        mov ah, 49h             ; free buffer
        mov es, di
        int 21h
        jc  error

        mov al, 00h             ; program succeeded
        jmp short fini

error:  mov al, 01h         ; program failed

fini:   mov ah, 4ch         ; exit to DOS
        int 21h

        end start
```

It is hard to see how this constitutes a DOS extender program: it makes no calls to a special API. That is largely the point of using a DOS extender.

But running under the DOS/16M debugger shows that something unusual is happening behind the scenes. After this program allocates a buffer with the standard MS-DOS function `Int 21H AH=48H`, we can examine the segment of the buffer returned in AX, which this program then stores in DI. Right away we can see something different from normal MS-DOS: the segment number is `90H`, which would *never* be returned by Function `48H` under normal circumstances. This number `90H` is a protected-mode selector. Instant-D provides a command that allows us to see the selector's physical base address, size, and other attributes:

```
>sel di
  90:  linear 1.ABFC.0, limit  FFF, data, NotRefd
```

This data segment is located above the first megabyte of memory, at physical address `1ABFC0H`. The last legal offset within the segment (the limit) is 4095 (`FFFH`). One important note: if you are running on a 386 with a control program like 386MAX or QEMM, the selector command does not display an absolute physical address, but instead shows a *linear* address based on 386 paging.

In real mode, the highest possible absolute address is `FFFFFH`, or at best `10FFEFH` if the A20 line is enabled with a utility like Microsoft's HIMEM.SYS. The absolute address `1ABFC0H` is way beyond the normal reach of MS-DOS.

After opening the file and calling `Int 21H AH=3DH` to read its contents into the buffer, we can examine the selector again:

```
>sel di
  90:  linear 1.ABFC.0, limit  FFF, data
```

The `NotRefd` attribute has gone away, indicating that this block of memory has now been accessed. We can also examine the segment directly:

```
>db di:0
  0090:0000 = 3B 20 20 20 20 20 20 20  ;
  0090:0008 = 66 69 6C 65 72 65 61 64  fileread
  0090:0010 = 2E 61 73 6D 0D 0A 3B 0D  .asm  ;
```

Thus, `Int 21H` Function `3FH` really has read from a file into an extended-memory buffer, without us doing anything besides relinking and postprocessing.

More Than One Int 21H

The DOS/16M debugger has a command to display entries from the protected-mode IDT:

```
>di 21
21: RM = 0EC4'06C3,  PM = 0070:076D
```

This shows that there are *two* interrupt service routines for `Int 21H`. The display interrupt command shows both the real-mode interrupt vector and the protected-mode interrupt gate. Instant-D uses the notation xxxx'xxxx to indicate real-mode addresses; xxxx:xxxx is used solely for protected-mode addresses. Here, the real-mode vector is the same as it was before DOS/16M took over. The protected-mode gate catches all `Int 21H` requests made by the protected-mode program. Some of these are serviced entirely in protected mode, and some are passed to the real-mode interrupt service routine (ISR).

Again, this is not all that different from the normal operation of MS-DOS. On a normal PC with a few TSRs loaded, every time a program issues an `Int 21H`, *several different* ISRs see it. The only difference here is that the first of these ISRs runs in protected mode.

Under what circumstances is an `Int 21H` request serviced entirely in protected mode, and when is it passed down to real-mode MS-DOS?

In the case of memory allocation (`AH=48H`), it depends on DOS/16M's allocation strategy, and on how much extended memory is available. If the DOS/16M strategy is "prefer extended," and if there is sufficient extended memory, the request can be handled entirely in protected mode. On the other hand, if the DOS/16M strategy is "force low," or if there is insufficient extended memory, the request must be subcontracted out to real-mode MS-DOS.

For file I/O, all `Int 21H` requests must be passed down to MS-DOS, because the DOS extender rightly knows nothing about the file system.

The DOS extender, however, cannot simply pass file I/O requests through to real mode. The example FILEREAD.ASM shows why: the buffer we want to use is located in extended memory, and MS-DOS doesn't really know how to access extended memory (a DOS extender just makes it look as if it did). In this program, even the tiny string containing the filename is located in extended memory, at `19B740H`:

```
>sel ds
  88:  linear 1.9B74.0, limit FFFF, data
```

In any event, even if the physical addresses *were* in low memory, DOS/16M would still have to translate the program's protected-mode selectors into real-mode segment numbers for MS-DOS.

When a protected-mode program requests a DOS file read (AH=3FH), on entrance to the DOS extender's Int 21H handler DS:DX holds a protected-mode address. After switching to real mode in preparation for resignaling the Int 21H, the selector in DS must be changed to a real-mode segment number (note that it can't be changed while the CPU is still in protected mode, since that would involve loading segment registers with values that are probably invalid in protected mode). After returning from the old Int 21H, but before putting the processor back in protected mode, DS must be restored. The offset in DX doesn't change. This is all that is required if DS happens to contain a selector that corresponds to an address in the first megabyte of memory.

But if DS corresponds to extended memory, it has no simple translation to a real-mode segment. A DOS extender maintains a buffer in low memory and, when reading from a file, will, behind the scenes, pass the low-memory buffer to MS-DOS for I/O, and then *copy* this buffer back into your extended-memory buffer. For writing to a file, the process is reversed: the DOS extender first copies your extended-memory buffer into low memory and then invokes the real-mode service. Your program is unaware that this is taking place. Since this process only occurs when extended-memory buffers are passed to real-mode services, and since the time required for file I/O is an order of magnitude greater than that required for the memcpy (REP MOVSB), there is hardly any performance penalty.

To summarize, a DOS extender allows MS-DOS to be called from protected mode by installing a protected-mode Int 21H handler, which does the following:

- puts the CPU into real mode
- performs various protected- to real-mode translations
- invokes the old real-mode Int 21H
- performs real- to protected-mode translation
- returns to protected mode.

Real-mode MS-DOS thinks it is talking to a normal program, and a protected-mode program thinks that MS-DOS knows how to handle its requests. The DOS extender sits in the middle, lying out of both sides of its mouth.

An In-Depth Look at the DOS/16M Toolkit

Because it examined an assembler program, the preceding discussion failed to answer one question: in the more realistic scenario of porting a C program to "Extended DOS," how is the real-mode output from the C compiler, particularly the real-mode code linked in from the C standard library, made to work in protected mode? This seems almost as magical as performing DOS calls with protected-mode selectors to extended-memory buffers.

For the most part, a 286-based protected-mode DOS extender relies on your existing real-mode tools: same compiler, same libraries, same linker. To explain how real-mode .OBJ, .LIB, and .EXE files are massaged for protected mode, we need to once more go over the three steps involved in producing the DOS/16M version of LIST.EXE from the real-mode LIST.OBJ produced by the standard Microsoft C compiler:

```
link /noe/map \16m\preload \16m\crt0_16m \16m\pml list,list,list;
 \16m\makepm list
 \16m\splice list.exe list.exp \16m\loader.exe
```

First, we linked in some additional .OBJ modules supplied with DOS/16M. 286-based DOS extenders generally support several different programming languages and compilers. DOS/16M supports Microsoft C, Turbo C, Watcom C, Lattice C, Microsoft Fortran, Zortech C++, Logitech Modula-2, as well as assembler (which, as we've seen, requires minimal support).

In the case of Microsoft C 5.1, the DOS/16M module CRT0_16M.OBJ replaces the default Microsoft startup code (CRT0.OBJ), and PRELOAD.OBJ provides placeholders for your program's GDT, IDT, and other selectors. Other modules are substitutes for the surprisingly few parts of the Microsoft function library that would generate a GP fault in protected mode.

For example, the `int86()` family of functions is important to low-level PC programming in C, but because the Intel `Int` instruction accepts only immediate values (`MOV AX, 5Ch` followed by `Int AX` is illegal: you must say `Int 5Ch`), Microsoft implements `int86()` by assembling the `Int` instruction while the program is running: three bytes of code are assembled on the stack and then CALL-ed. But this constitutes executing data, which is illegal in protected mode.

Also, when using `int86x()` or `intdosx()`, it is very easy to load a segment register with an uninitialized segment number: in real mode, an unused bogus value in ES doesn't cause a problem, but in protected mode, the simple act of loading an invalid value into ES, even if you don't intend to use it, instantly causes a GP fault.

For these reasons, one of the modules provided by DOS/16M replaces Microsoft's version of `int86x()` with one that works in protected mode. The replacement looks up the interrupt in the IDT (which in DOS/16M-based programs is always at segment `10H`), pushes the interrupt handler's address on the stack, and does an IRET. Before loading segment registers, the `int86x()` and the `intdosx()` functions use the Intel `LAR` instruction to make sure the segment registers contain valid values.

With this fix, `int86x()` can work in protected mode. And this is one of the more substantial changes that a DOS extender needs to make to a real-mode C library. It illustrates the minimal changes needed to get even low-level PC code running under a DOS extender, and is a good indication of the type of changes you may need to make to your own code. The low-level DOS and BIOS interface routines provided with Microsoft C 5.1, such as `_dos_findfirst()` and `_bios_disk()`, operate in protected mode without modification.

The few DOS/16M object modules, together with real-mode libraries and your program's object modules, can be passed to a DOS linker such as LINK.EXE or PLINK.

There is one problem with DOS/16M's use of the DOS linker: what if your program's code exceeds one megabyte? DOS linkers can't handle this much code, except as overlays. DOS/16M therefore allows you to link a huge executable as though it were using overlays. This is merely to make the program acceptable to LINK; the overlay structure is flattened again by MAKEPM.

Another way to avoid the one-megabyte limitation of LINK is to use Phar Lap's LinkLoc, which can link programs directly for 286 protected mode. When using LinkLoc with DOS/16M, there is no need to use MAKEPM, since LinkLoc already assigns protected-mode selectors for addresses.

MAKEPM.EXE: A Postprocessor

If you are using the DOS linker, the next step is to run the DOS/16M MAKEPM utility, which prepares executables for protected mode. Its screen output, shown in Figure 4-1, shows what MAKEPM does.

The MAKEPM display says that it has relocated real-mode segment references to protected-mode selectors. If you were to examine the file LIST.EXE before passing it to MAKEPM, you might find code such as:

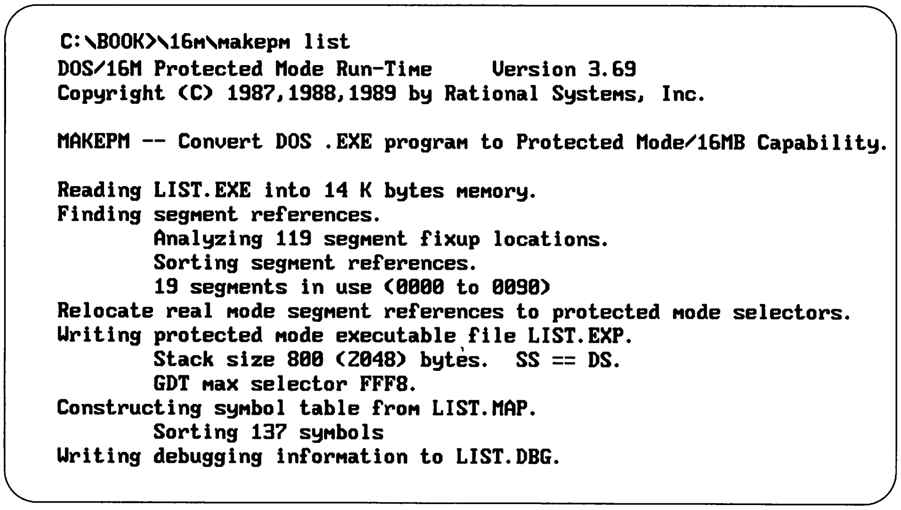
```
4EF4:0091        CALLF 4EF4:0434
```

After running MAKEPM, in the file LIST.EXP, this same piece of code becomes:

```
0080:0091        CALLF 0080:0434
```

Using the relocation table in LIST.EXE, MAKEPM locates all inter-segment references (segment fixup locations) and, without altering the offset portion, patches the segment to refer instead to a protected-mode selector.

Figure 4-1: MAKEPM screen output.



```
C:\BOOK>\16M\makepm list
DOS/16M Protected Mode Run-Time     Version 3.69
Copyright (C) 1987,1988,1989 by Rational Systems, Inc.

MAKEPM -- Convert DOS .EXE program to Protected Mode/16MB Capability.

Reading LIST.EXE into 14 K bytes memory.
Finding segment references.
        Analyzing 119 segment fixup locations.
        Sorting segment references.
        19 segments in use (0000 to 0090)
Relocate real mode segment references to protected mode selectors.
Writing protected mode executable file LIST.EXP.
        Stack size 800 (2048) bytes.  SS == DS.
        GDT max selector FFF8.
Constructing symbol table from LIST.MAP.
        Sorting 137 symbols
Writing debugging information to LIST.DBG.
```

The number 4EF4H in the real-mode executable depends entirely on where this program is loaded in memory. Loading on a different machine, with a different version of DOS or with a different mix of TSRs running, would result in a different segment number. But in the protected-mode .EXP file, the selector 0080 will never change. Selectors, remember, are logical rather than physical units. The physical base address for the selector may well change from one run of the program to the next, but the selector itself remains fixed. This extra level of indirection greatly eases debugging.

Since MAKEPM takes an already compiled and linked executable and transforms it, it is a *postprocessor*, a mechanical translator.

The Microsoft linker has an option, /FARCALLTRANSLATION, which relates to the operation of MAKEPM. With far-call translation, when the linker sees an inter-segment reference in which the source and target are the same, such as:

```
    4EF4:0091        CALLF 4EF4:0434
```

it is able to translate the far/long call into a near/short call:

```
    4EF4:0091        NOP
    4EF4:0092        PUSH CS
    4EF4:0093        CALL 0434
```

This is no longer a segment fixup location, so MAKEPM has nothing to do here. The resulting protected-mode executable runs faster since this block of code is no longer loading a value into the CS register. Loading segment registers—even redundantly loading them with the same values as their current value—is expensive in protected mode, so far-call translation can be a useful protected-mode optimization (though explicitly using the *near* keyword is even better).

MAKEPM has many command-line options to alter the run-time configuration of a DOS/16M program. These can give a program more memory to run, speed up its performance, or help with debugging. Recall that if we simply ran MAKEPM on the LIST program, without any options, the resulting executable was able to allocate 1696K of memory on a 2-megabyte Compaq 286. By tweaking MAKEPM, we can get 1772K, almost another 100K:

```
        \16m\makepm -gdt 0x400 -buffer 2048 list
```

By default, MAKEPM creates a 64K GDT for your program. This program doesn't use the transparent selectors that account for much of the GDT's size, so we use the -gdt switch to allocate a smaller GDT, freeing up more memory for the linked list. Furthermore, DOS/16M, by default, creates an 8K low-memory buffer for I/O transfer with MS-DOS. A program that does a lot of file I/O would get better performance by making the buffer larger. But our sample program does no file I/O whatsoever, so we can reduce the buffer size. In a genuine program, this would be a time/space trade-off.

Another MAKEPM option, -mfl, forces all memory allocation to take place in the lower 640K. This is extremely useful if you want to port an application to protected mode solely for debugging or protection purposes. By using the -mfl option from MAKEPM, together with the DOS/16M function call `D16MemStrategy (MTransparent)`, and occasional calls to `D16RealPtr()`, you can get the advantages of hardware-enforced protection, even for programs that don't require extended memory.

SPLICE.EXE, Packages, and Transparency

In order to take the EXP file produced by MAKEPM and run it from the DOS command line, you must use the DOS/16M SPLICE utility. In earlier releases of DOS/16M, SPLICE merely took your protected-mode program and DOS/16M's kernel/loader (LOADER.EXE) and merged them. But since the kernel/loader is a separate executable, this opened up the possibility for splicing in alternate loaders. For example, DOS/16M provides WINLOAD.EXE, which can be used to create protected-mode executables for Windows/286 (they won't load under Windows/386, though).

SPLICE has evolved into a utility for merging several different .EXP files, together with a protected-mode loader, into one protected-mode MS-DOS executable. Since DOS/16M, like other DOS extenders, does not provide concurrent tasks, what does it mean when several different .EXP files are run together? One of the .EXP files is your application program; the others are what Rational Systems calls *packages*. In addition to such add-ins as the VM and overlay managers, packages allow transparent access to those software interrupts which are otherwise unsupported by the DOS/16M kernel.

In order to provide transparent protected-mode access to a real-mode service, the DOS/16M kernel must know about the service. Thus, DOS/16M knows about the individual functions provided by `Int 21H`. But what if you want to use some other real-mode service, such as the Microsoft mouse driver (`Int 33H`) or NetBIOS (`Int 5CH`), knowledge of which is not hard-wired into the DOS/16M kernel? You could use the DOS/16M function library to non-transparently manage protected- to real-mode translation each time you use the software interrupt, but a better way would be to use the DOS/16M function library to write a separate application, a package, which takes over the interrupt and provides other applications transparent access to it in the same way that transparent access is provided for `Int 21H`.

DOS/16M currently comes with packages for the mouse, for those `Int 10H` functions not supported by the DOS/16M kernel (for example, palette manipulation and character generator), and for NetBIOS.

Packages enable DOS/16M capabilities to be extended without adding overhead to the kernel. Future releases will enable you to build your own packages, and build applications that consist of several cooperating programs that share the same address space. Note that packages are *not linked* together with your application: they are *spliced* together after link-time and after MAKEPM postprocessing. Thus, the memory model, compiler, and so on, used to build the package are irrelevant to your application.

DOS/16M packages are another instance of the DOS extender emphasis on, above all, transparency. James Smith, in *The IBM PC/AT Programmer's Guide*, observes that "Something is *transparent* if it is really there but seems not to be." Somewhat reminiscent of early MS-DOS's relationship to CP/M, DOS extenders want you to have to break as few of your habits, bad or otherwise, as possible. DOS extenders represent a clean break from the 8088, without breaking from MS-

DOS. Contrast this to OS/2, which requires a totally different programming world view.

Both approaches—the "Extended DOS" emphasis on transparency, and the OS/2 emphasis on making a complete break with the past—make perfect sense.

BANNER.EXE

At this point, we have this strange beast, a protected-mode executable that runs from real-mode MS-DOS. When we start the program from the DOS command line, it displays a brief copyright message:

```
C:\BOOK>list
DOS/16M Protected Mode Run-Time     Version 3.69
Copyright (C) 1987,1988,1989 by Rational Systems, Inc.
```

This is useful during development so you know which version of the executable you are running (though it is usually obvious: the DOS/16M version is the one that doesn't produce *Fatal error: out of memory* messages), but your users probably don't need to see this message. For example, Lotus 1-2-3 Release 3 does *not* display this message. To turn the message off, you can use the DOS/16M BANNER utility:

```
banner list off
```

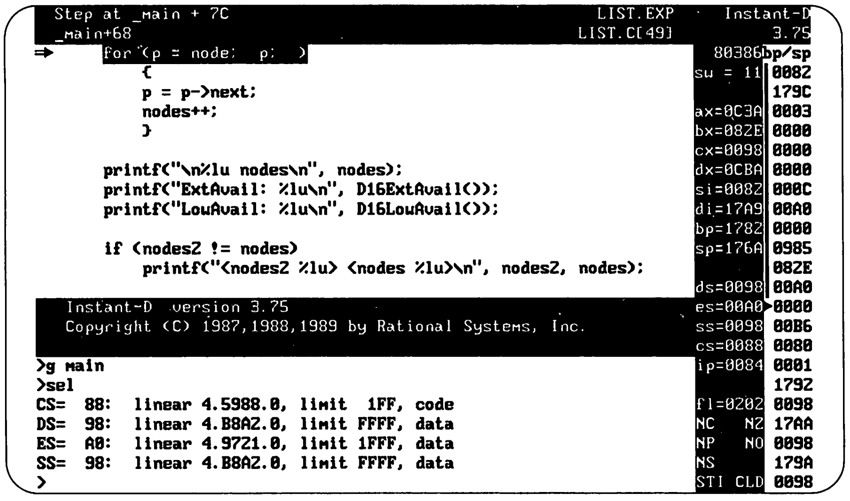
D.EXE

Using the DOS/16M debugger with an assembler program, as we did earlier, does not show D.EXE in its true light, since this is a *source-level* debugger (see Figure 4-2). In addition to the usual features found in CodeView and Turbo Debugger, and the *selector* and *display interrupt* commands we've already used, Instant-D provides commands such as those shown in Table 4-1.

Table 4-1: Instant-D commands.

Command	Description
absolute	Display absolute physical address for selector and offset
cpu	Display the MSW on 286 machines, or CR0 on 386s
files	List all open files
freelist	Display DOS/16M extended-memory free list
imr	Display the Interrupt Mask Register
opt	Set DOS/16M run-time options
snap	Record top of stack to a file
where	Displays absolute address, source file, and any protected-mode aliases for a symbol

Figure 4-2: Instant-D, the DOS/16M source level debugger.



```
  Step at _main + 7C                                          LIST.EXP  ·  Instant-D
  _main+68                                                   LIST.C[49]        3.75
=>      for (p = node;  p;  )                                         80386bp/sp
            {                                                         sw = 11 0082
            p = p->next;                                                      179C
            nodes++;                                                  ax=0C3A 0003
            }                                                         bx=082E 0000
                                                                      cx=0098 0000
        printf("\n%lu nodes\n", nodes);                               dx=0CBA 0000
        printf("ExtAvail: %lu\n", D16ExtAvail());                     si=0082 000C
        printf("LowAvail: %lu\n", D16LowAvail());                     di=17A9 00A0
                                                                      bp=1782 0000
        if (nodes2 != nodes)                                          sp=176A 0985
            printf("<nodes2 %lu> <nodes %lu>\n", nodes2, nodes);              082E
                                                                      ds=0098 00A0
   Instant-D  version 3.75                                            es=00A0>0000
   Copyright (C) 1987,1988,1989 by Rational Systems, Inc.             ss=0098 00B6
                                                                      cs=0088 0080
>g main                                                               ip=0084 0001
>sel                                                                          1792
CS=  88:  linear 4.5988.0, limit  1FF, code                           fl=0202 0098
DS=  98:  linear 4.B8A2.0, limit FFFF, data                           NC   NZ 17AA
ES=  A0:  linear 4.9721.0, limit 1FFF, data                           NP   NO 0098
SS=  98:  linear 4.B8A2.0, limit FFFF, data                           NS      179A
>                                                                     STI CLD 0098
```

Several other Instant-D commands are crucial in porting programs to protected mode. The *sampler* command, for instance, controls Instant-D's built-in performance monitoring facility. This facility can help you radically improve the performance of your protected-mode program. The *backtrace* command is useful when your program GP faults. By tracing back along the call chain, you can find out where your program went wrong. (Instant-D also implements a *visual backtrace* in which Ctrl-PgUp displays the caller's code.)

In addition to displaying real-mode addresses as xxxx'xxxx, and protected-mode addresses as xxxx:xxxx, Instant-D also accepts input in this form (for example, `bp 1234'5678` sets a breakpoint on that real-mode address), and &xxxxxx is used to refer to absolute addresses.

A final important feature of Instant-D is the ability to invoke it from within your program. For example:

```
    if (fp == NULL)
        D16CallDebug("backtrace");   // how did I get here?
```

This programmatic interface to the debugger is such a good idea that it is surprising more debuggers don't provide it.

PMINFO.EXE and RMINFO.EXE

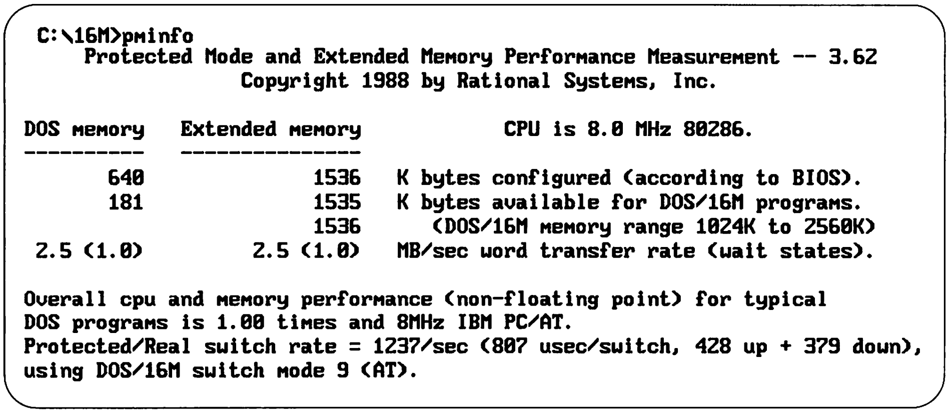
The DOS/16M toolkit includes a number of useful utilities that you can distribute to customer sites along with your executable.

PMINFO is somewhat like Norton's SI, for protected mode: in addition to the amount of extended memory available to DOS/16M programs and the memory transfer rate, PMINFO measures the all-important protected/real switch rate—both the maximum number of round-trip switches that occur per second, and the number of microseconds required for one switch, broken into its real-to-protected (up) and protected-to-real (down) components—and indicates which of several different switch techniques DOS/16M will use on the particular machine.

Figures 4-3 and 4-4 show PMINFO output, first for an 8 MHz IBM AT, then for a 16 MHz Compaq 386. This shows that, as mentioned earlier, an IBM AT (286) can switch about 1,200 times per second, while a Compaq 386 can switch over 7,000 times per second.

On a 386 computer, if Qualitas's famous memory manager 386-to-the-Max is loaded, PMINFO holds a surprise: the switch rate drops from over 7,000 round trips per second, to around 2,000—not much better than a 286.

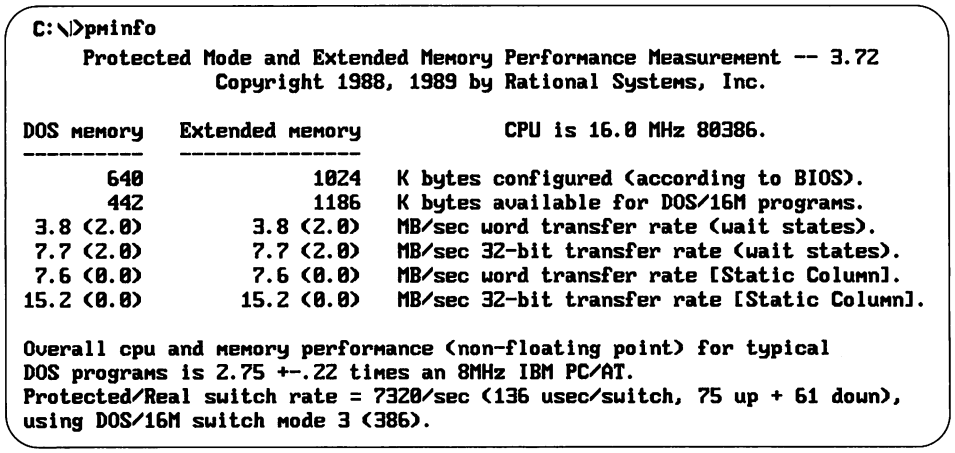
Figure 4-3: PMINFO display for an 8 MHz IBM AT.



```
 C:\16M>pminfo
     Protected Mode and Extended Memory Performance Measurement -- 3.62
                    Copyright 1988 by Rational Systems, Inc.

DOS memory   Extended memory            CPU is 8.0 MHz 80286.
----------   ---------------
      640              1536   K bytes configured (according to BIOS).
      181              1535   K bytes available for DOS/16M programs.
                       1536     (DOS/16M memory range 1024K to 2560K)
 2.5 (1.0)        2.5 (1.0)   MB/sec word transfer rate (wait states).

Overall cpu and memory performance (non-floating point) for typical
DOS programs is 1.00 times and 8MHz IBM PC/AT.
Protected/Real switch rate = 1237/sec (807 usec/switch, 428 up + 379 down),
using DOS/16M switch mode 9 (AT).
```

Since PMINFO is itself a DOS/16M program, it can be used as a basic test of a configuration's ability to run your program. If PMINFO fails (possible messages include *Protected mode failure, Not enough extended memory, Computer must have 80286 or 80386 CPU,* and *Protected mode requires VCPI within Virtual 8086*), the solution is often extremely simple, such as getting the new VCPI-compatible version 4.0 of CEMM (Compaq expanded-memory manager). The issue of software incompatibilities is discussed in Chapter 8.

Figure 4-4: PMINFO display for a 16 MHz Compaq 386.



```
C:\|>pminfo
     Protected Mode and Extended Memory Performance Measurement -- 3.72
                  Copyright 1988, 1989 by Rational Systems, Inc.

DOS memory   Extended memory             CPU is 16.0 MHz 80386.
----------   ---------------
       640              1024   K bytes configured (according to BIOS).
       442              1186   K bytes available for DOS/16M programs.
 3.8 (2.0)         3.8 (2.0)   MB/sec word transfer rate (wait states).
 7.7 (2.0)         7.7 (2.0)   MB/sec 32-bit transfer rate (wait states).
 7.6 (0.0)         7.6 (0.0)   MB/sec word transfer rate [Static Column].
15.2 (0.0)        15.2 (0.0)   MB/sec 32-bit transfer rate [Static Column].

Overall cpu and memory performance (non-floating point) for typical
DOS programs is 2.75 +-.22 times an 8MHz IBM PC/AT.
Protected/Real switch rate = 7320/sec (136 usec/switch, 75 up + 61 down),
using DOS/16M switch mode 3 (386).
```

Sometimes, though, PMINFO detects a hardware incompatibility. The BIOS in a few AT clones was designed solely for real-mode compatibility; no one was thinking about protected mode at the time. These clones are incompatible in their handling of the protected-to-real-mode shutdown switch as well as in their handling of the A20 line, and will have the same problem running OS/2 as they have running DOS/16M or OS/286. For DOS/16M, there is often a simple workaround that uses `SET DOS16M=` in the DOS environment.

The RMINFO utility is similar to PMINFO, but stops just short of switching into protected mode. RMINFO reports the presence of other extended-memory users (such as VDISK, XMS, and VCPI), indicates whether DOS/16M would use triple faulting to switch back to real mode from protected mode, indicates how DOS/16M will handle the A20 line while switching modes, and also tells whether any DOS/16M TSRs are loaded (*DOS/16M is already running*).

Isn't there any work involved?

At this point it should be clear that most real-mode DOS code will work, *as is*, under a 286-based protected-mode DOS extender. All you do is add a few extra .OBJ modules, link, and run the executable through a mechanical translator.

The best way to port a DOS program to a 286-based DOS extender is *just to do it*. Construct a new batch file or make file, link the DOS/16M-provided .OBJs with your present .OBJ modules, and see if it runs. This is the point at which the typical software engineer asks, "Where's the work?"

The likelihood is that such a simply ported program will run for a while, but will eventually break one of protected mode's rules, and be terminated with a protection violation (GP fault). These rules include:

- Don't peek or poke memory not owned by your task.
  Example: `*((unsigned long far *) 0x0000046C)`
- Don't execute data.
  Example: Microsoft's `int86x()`
- Don't move data into code segments.
  Example: `MOV CS:foo, AX`
- Don't misuse segment registers.
  Example: using ES as a scratchpad register
- Don't perform real-mode address arithmetic.
  Example: pointer normalization

You now have work to do: fixing the code so that it runs in protected mode. In practice, two areas tend to require the most work when you convert to a DOS extender: storing data in a code segment, and performing address arithmetic.

Impure Code

Protected mode forbids storing into a code segment. Normally, this is a wonderful feature: it prevents overwriting-code bugs. Furthermore, making sure that a code segment contains only executable code (pure code) allows it to be shared by tasks in a multitasking environment.

But what if your program *deliberately* stores data in a code segment?

It seems that almost any .ASM file you pick will contain at least one instance of this. Now, using a CS: override in the *source* for a `MOV` does *not* cause a GP fault, because reading from a code segment is permissible:

```
        mov ax, cs:request
```

But using a CS: override in the *destination* for a `MOV`, writing into a code segment, definitely causes a GP fault:

```
mov cs:request, ax
```

Unfortunately, the use of CS as a storage area is not always so explicit. For example, assembler that uses the SEGMENT and ASSUME directives can easily construct "impure code" without using an explicit CS: override:

```
_TEXT  segment public 'CODE'
       assume CS:_TEXT, DS:_TEXT
;; ...
       mov request, ax
```

Code that uses the new directives, such as DOSSEG and .CODE, is less likely to contain impure code, but is still possible:

```
        286
        dosseg
        model large
        .code
request dw      ?
start:
        mov request, 1234h
        end start
```

Fortunately, the Microsoft assembler (MASM) has an option to check for impure code. If you put the `.286` directive at the top of a source file, and use the option -p when you assemble, MASM locates all lines containing an impure memory reference. Borland's Turbo assembler (TASM) provides the same feature, with the warning *CS override in protected mode.*

Having located all such places in your code, you have to ask *why* data is being stored in a code segment. Often, it's simply unnecessary, and can easily be fixed. The previous example could be rewritten so that it works in both protected and real mode, simply by putting the data where it belongs, in the data segment:

```
        .286
        dosseg
        .model large
        .data
request dw      ?
        .code
start:
        mov request, 1234h
        end start
```

Sometimes, however, it really does make sense to keep data in a code segment. For example, an interrupt service routine (ISR) generally *must* have at least one variable accessible from CS, since this is the only segment register whose value is known when the ISR is invoked.

The easiest way to make such code work in a DOS extender is to read the data from the code segment with a CS: override, and write data into the code segment using an "alias" selector.

If you are at all familiar with OS/2 programming, you may know about the function `DosCreateCSAlias()`, which creates a selector that points to the same absolute address as a given data selector, but is executable.

DOS extenders also have "alias" selectors and, in addition to the function `D16SegCSAlias()`, the DOS/16M library (which we'll discuss in more detail later) also has `D16SegDSAlias()`, which creates a selector that points to the same absolute address as a given code selector, but is writable. That is exactly what we need here.

To avoid the overhead of allocating and cancelling the alias each time we need to use it, which will probably be in an ISR, we create the alias during initialization and store the alias in the segment to which the alias points. At invocation, we read the alias out of CS:alias and then move the alias into DS (remembering, of course, to save and restore the previous value in DS). Before exiting, we cancel the selector.

The two DOS/16M functions used here expect to be called from a high-level language, and have the following C prototypes:

```
void far *D16SegDSAlias (void (far *pm_codeptr)());
int D16SegCancel (void far *pm_dataptr);
```

The following example is in assembler, because while this issue has frequently come up in assembler subroutines for C programs (for example, in the Greenleaf telecommunications library), one usually has no occasion to call these functions from C. To call the DOS/16M functions from assembler, the parameters are pushed on the stack and cleared off afterwards:

```
;       a.asm
        .286
        public _init, _subr, _fini

ifdef DOS16M
        extrn _D16SegDSAlias : far
        extrn _D16SegCancel : far
endif
```

```
A_TEXT  segment public 'CODE'
        assume CS:A_TEXT, DS:ATEXT

foo     dw  ?                ; the data we're interested in
alias   dw  ?                ; the alias for code segment

_init   proc far
ifdef DOS16M
        push bx
        push A_TEXT
        push 0
        call far ptr _D16SegDSAlias
        add sp, 4            ; alias SEG is now in DX
        push ds              ; remember to save DS
        mov ds, dx           ; move alias for CS into DS
        mov alias, dx        ; store alias sel in alias seg!
        pop ds               ; restore DS
        pop bx
else
        push ds
        mov ax, A_TEXT       ; for portability, pretend we need
        mov ds, ax           ; an alias in real mode too
        mov alias, ax
        pop ds
endif
        ret
_init   endp

_subr   proc far
        push ds
        mov ax, cs:alias     ; okay to read from CS
        mov ds, ax           ; put alias for CS into DS
        mov foo, 1234h       ; do the work!
        pop ds
        ret
_subr   endp


_fini   proc far
ifdef DOS16M
        push bx
        push word ptr cs:alias
        push 0
        call far ptr _D16SegCancel
        add sp, 4
        pop bx
endif
        ret
```

```
_fini   endp

A_TEXT  ends

        end
```

This example could be called from a C program such as the following (though remember that `subr()` would probably be part of an interrupt handler):

```
/* c.c */
extern void init(void), subr(void), fini(void);
main()
{
    int i;
    init();
    for (i=0; i<10000; i++)     // simulate real work
        subr();
    fini();
}
```

For the functions `D16DSAlias()` and `D16SegCancel()`, we need to link with the DOS/16M function library (which is provided as C source code):

```
if not exist \16m\dos16lib.obj cl -AL -Ox -Gs2 -c \16m\dos16lib.c
masm -ml -DDOS16M a;
cl -AL -Ox -c c.c
link /noe \16m\msc5.1\preload \16m\msc5.1\crt0_16m c a \
    \16m\msc5.1\pml \16m\dos16lib,c,c;
makepm c
splice c c \16m\loader.exe
```

Notice that the C module is unchanged for protected mode and that the overhead of creating and cancelling the alias is confined to infrequently-called parts of the program.

The ability to create a DS alias is not unique to DOS/16M. OS/286 has a similar "create data window" function in its "extended segment services" (`Int 21H AX=E801H`). Note, however, that in OS/286, every code selector by default comes with its own matching data window.

Address Arithmetic

In addition to impure code, the other issue that comes up repeatedly when you port to protected mode is address arithmetic. Aspects of real-mode address arithmetic include adding a length to a pointer to compute the next pointer, performing a carry from the offset to the segment portion of an address, and relying on a simple conversion from segment:offset pointers to absolute physical addresses.

Some large applications implement their own dynamic code loaders, virtual memory managers, or overlay schemes; some object-oriented programs support "persistent objects" which can be written out to disk and later read back in (perhaps on a different machine). All of these use address arithmetic to perform address relocations or fixups. When reading objects (code or data) in from disk, pointers can be computed simply by adding an offset to a base "load address." Pointers are stored on disk as offsets from location zero.

It is often said that protected mode forbids any form of address arithmetic. After all, the selectors in addresses returned by `malloc()` are just "magic cookies," bearing no relation to their underlying physical addresses.

In fact, certain forms of address manipulation are the same as in real mode, and, should you need it, there is even the potential for a kind of protected-mode address arithmetic.

First, the useful `FP_SEG()`, `FP_OFF()`, and `MK_FP()` macros all still work in 16-bit protected mode (though they are totally different in 32-bit protected mode):

```
typedef unsigned WORD;
typedef unsigned long DWORD;
typedef void far *FP;
#ifdef LVALUE
#define FP_SEG(fp)  (*((WORD *)&(fp) + 1))
#define FP_OFF(fp)  (*((WORD *)&(fp)))
#else
#define FP_OFF(fp)  ((WORD)(fp))
#define FP_SEG(fp)  ((WORD)((DWORD)(fp) >> 16))
#endif
#define MK_FP(a,b)  ((FP)(((DWORD)(a) << 16) | (b)))
```

For example:

```
extern char far *foo;
WORD sel = FP_SEG(foo);
```

or:

```
extern WORD _psp;
struct PSP far *psp_ptr = MK_FP(_psp, 0);
```

Here, however, the similarity to real mode ends. For example, the following, which uses the "LVALUE" form of the macros, makes no sense in protected mode:

```
FP_SEG(foo)++;
```

In real mode, adding 1 to a segment value describes a physical location one paragraph (16 bytes) higher in memory, and is equivalent to adding `10H` (16) to the segment offset. In protected mode, there is no such carry from the offset into the segment, and while adding 1 to a selector is permissible (the preceding code does *not* cause a GP fault), *dereferencing* the resulting pointer definitely does not produce the expected results, and on some systems (such as OS/2) even generates a protection violation.

Code written for real mode often relies on the ability to convert between a real-mode pointer and its underlying physical address:

```
#define PTRTOABS(fp)     (DWORD) ((FP_SEG(fp) << 4) + FP_OFF(fp))
#define ABSTOPTR(abs)    (MK_FP((DWORD) (abs) >> 4, (abs) & 0x0F))
```

It was mentioned earlier that real-mode segments overlap, so that `0000:046C` and `0040:006C` are equivalent. To reliably compare two real-mode pointers for equality, the pointers thus must be "normalized." The preceding macros can be bundled together in real mode to reduce pointers to canonical form, so that their offset is always less than 16:

```
#define NORMALIZE(fp)      (ABSTOPTR(PTRTOABS(fp)))
#define CMP_PTR(fp1,fp2)   (NORMALIZE(fp1) == NORMALIZE(fp2))
```

But this doesn't make any sense in protected mode. Instead, you can think of protected-mode pointers as already "normalized." To compare two pointers for equality, just compare them:

```
#define CMP_PTR(fp1,fp2)   ((fp1) == (fp2))
```

To see if one pointer is located within the same segment as another pointer, real-mode code generally manipulates their underlying absolute addresses. But since the underlying absolute address is of almost no importance in protected mode, and since the only meaningful item you can look at is the selector, this comparison becomes:

```
#define SAME_SEG(fp1,fp2)  (FP_SEG(fp1) == FP_SEG(fp2))
```

In short, certain operations that are required in real mode because of overlapping segmentation are simply unnecessary in protected mode.

How much does this buy us? A fair amount, but there is still a problem. The MS-DOS memory-allocation function (`Int 21H AH=48H`) can allocate blocks of memory larger than 64K. This function is passed the number of 16-byte *paragraphs* to be allocated, and returns the *initial* segment of the entire block. With a 16-bit parameter, a one-megabyte block can potentially be allocated. In real mode, this is a fiction, but in protected mode it is a very real possibility. In either mode, with a 16-bit offset, any given segment or selector can at most address 64K. How, then, do you jump from one segment or selector to the next?

If you use your compiler's huge model, the segment arithmetic is taken care of for you. But programs that do their own relocations and fixups don't use huge model; they do the arithmetic themselves. Even some programs that do use the slower huge model rely on the ability to convert between huge and far pointers.

An example of code that relies on the ability to jump from one memory segment to the next, using some fixed increment, is Microsoft's own `heapwalk()` and `heapchk()` functions. As is, these functions do not work in a program prepared for DOS/16M. But they can be made to work by linking in a tiny object module, HDIFF.OBJ, which comes from the OS/2 Microsoft C libraries, and which is included with the DOS/16M toolkit.

So what increment is used in DOS/16M to jump from one 64K block to the next? The following small program investigates this. It calls the Microsoft C function `_dos_allocmem()`, which is a C front-end to `Int 21H AH=48H`. The program allocates `A000H` paragraphs (640K), and then loops 10 times, each time incrementing the segment portion of a pointer and printing out the base address and limit (size - 1) of the segment. Two DOS/16M functions, `D16SegLimit()` and `D16AbsAddress()`, are called to get this information. The program uses the `FP_SEG()` macro from Microsoft's dos.h, and prints the far pointer using the `printf()` `"%Fp"` mask:

```
#include <stdio.h>
#include <dos.h>
#include "dos16.h"

main()
{
    char far *fp = NULL;
    unsigned seg;

    _dos_allocmem(0xA000, &seg);
    for (FP_SEG(fp)=seg; FP_SEG(fp) < seg+10; FP_SEG(fp)++)
        printf("FP=%Fp LIMIT=%04X ABS_ADDR=%08lX\n",
            fp, D16SegLimit(fp), D16AbsAddress(fp));
    _dos_freemem(seg);
}
```

The output of this DOS/16M program indicates there really is a protected-mode segment arithmetic, but that it is quite different from real mode. Even the segment numbers themselves are different from anything seen in real mode:

```
FP=00B0:0000 LIMIT=FFFF ABS_ADDR=00150CF0
FP=00B1:0000 LIMIT=FFFF ABS_ADDR=00150CF0
FP=00B2:0000 LIMIT=FFFF ABS_ADDR=00150CF0
```

```
FP=00B3:0000 LIMIT=FFFF ABS_ADDR=00150CF0
FP=00B4:0000 LIMIT=FFFF ABS_ADDR=00150CF0
FP=00B5:0000 LIMIT=FFFF ABS_ADDR=00150CF0
FP=00B6:0000 LIMIT=FFFF ABS_ADDR=00150CF0
FP=00B7:0000 LIMIT=FFFF ABS_ADDR=00150CF0
FP=00B8:0000 LIMIT=FFFF ABS_ADDR=00160CF0
FP=00B9:0000 LIMIT=FFFF ABS_ADDR=00160CF0
```

The program is able to increment the segment portion of the address using `FP_SEG(fp)++`, but what does that mean? The physical address `150CF0H`, first of all, bears no relation to the segment numbers and, furthermore, doesn't even change until we get to segment `B8H`. When the absolute address does change, it jumps by 64K.

In contrast, a real-mode version of this program might produce output something like this:

```
FP=4309:0000 LIMIT=0010 ABS_ADDR=00043090
FP=430A:0000 LIMIT=0010 ABS_ADDR=000430A0
FP=430B:0000 LIMIT=0010 ABS_ADDR=000430B0
      .
      .
      .
FP=5309:0000 LIMIT=0010 ABS_ADDR=00053090
FP=530A:0000 LIMIT=0010 ABS_ADDR=000530A0
```

In real mode, adding one to the segment value takes us to the next paragraph. Thus, there are no abrupt quantum jumps as in protected mode, but it does require a large increment of `1000H` (4096) to move from one 64K block to the next. This just reflects the tiny address space available in real mode.

That it has taken eight segment numbers for the physical base address to change in protected mode is no accident. This will be explained when we examine the protected-mode data structures in detail using Instant-C, but the bottom three bits of a protected-mode selector have special meaning. To ignore these bits we must increment a selector by 2^3, or 8. This is the *minimum* increment to move from one 64K block to the next in a protected mode "huge" allocation. While DOS/16M uses the minimum increment of 8, OS/2, for example, uses an increment of 16.

OS/2 provides a function, `DosGetHugeShift()`, which returns a left-shift count which, applied to the number 1, produces the segment-arithmetic increment. In current releases of OS/2, `DosGetHugeShift()` returns 4 (1 << 4 = 16). Since `DosGetHugeShift()` is a "Family API" function for use in dual-mode executables, we could write our own DOS/16M-specific version. This uses the DOS/16M function ispm() to make sure the machine is in protected mode:

```
USHORT APIENTRY DosGetHugeShift(PUSHORT puShift)
{
    *puShift = (_is_pm()) ? 3 : 12;
    return 0;
}
```

We can apply all this to the problem of address relocation. If you have real-mode code that stores pointers on disk with something like:

```
extern void far *base;
DWORD disk_ptr = PTRTOABS(fp) - PTRTOABS(base);
```

(where base is the starting position for the workspace that is being saved to disk), and then reads them back in with something like:

```
FP ptr = ABSTOPTR(base + disk_ptr);
```

(where base is now the new "load address") in protected mode you will instead have to save pointers on disk in a way that preserves the distinction between the segment and offset:

```
DWORD disk_ptr;
WORD huge_shift;
DosGetHugeShift(&huge_shift);
disk_ptr = MK_FP((FP_SEG(fp) - FP_SEG(base)) >> huge_shift,
     FP_OFF(fp));
```

For example, if the base address is 00A8:0000 and we are storing the pointer 00C0:FEDC to disk, in real mode this could be stored in normalized form as 27EDCH, but in protected mode, if the huge_shift is 3, we would store 3FEDCH. To read this back in, we would use:

```
FP ptr = MK_FP(FP_SEG(base) + (FP_SEG(disk_ptr) << huge_shift),
    FP_OFF(disk_ptr));
```

If you are porting a program that does fixups or address relocation to protected mode, there are actually several possible solutions.

First, you can see if the code is still necessary under protected mode. Perhaps you have implemented your own overlay manager. After all, you may be using overlays or dynamic linking just to get around the DOS 640K limit. But remember, DOS/16M and OS/286 provide big memory but not virtual memory. Don't be too hasty to get rid of overlays, just because *your* computer has 4 megabytes. Users may have only 1 or 2 megabytes. But if you know that your users will have sufficient physical memory on their computers, you may be able to eliminate overlays entirely. Or you may want to use DOS/16M's overlay manager or VM manager instead.

Second, you can avoid use of `Int 21H AH=48H` to allocate more than 64K at a time. Code that performs address fixups might then require two passes, however.

Third, you can rely on the structure of protected-mode selectors and use the "huge shift."

Limits to Transparency

Aside from real-mode coding practices that must be changed for protected mode, a few other strictly MS-DOS programming practices might not work transparently under a DOS extender:

- using unsupported software interrupts
  Example: `Int 5CH` (NetBIOS)
- using undocumented DOS calls
  Example: `Int 21H AH=52H` ("get DOS pointer table")
- using unsupported absolute memory locations
  Example: `*((unsigned long far *) 0x46C)`

Don't be alarmed by this list. *Unsupported* simply means that the DOS extender doesn't provide *transparent* access. You can still do all this, but you do have to take some extra steps.

For example, we mentioned earlier that if the sample program LIST.C had been coded slightly differently, it could have caused a protection violation. Instead of calling the C `time()` function, the code could have slightly more efficiently—and considerably less portably—peeked at the BIOS data location `46CH`:

```
#define TICKS()   *((unsigned long far *) 0x46c)  // 0000:046C
#define SECONDS() (TICKS() / 18)
// ...
unsigned long t1 = SECONDS();
```

If a program executes this code in protected mode, it is terminated with a GP fault (`Int ODH`):

```
C:\BOOK>list
DOS/16M Protected Mode Run-Time    Version 3.73
Copyright (C) 1987,1988,1989 by Rational Systems, Inc.

DOS/16M: Unexpected Interrupt=000D  at 0088:0049
code=0000 ss=00A0 ds=00A0 es=0000
ax=0200 bx=046C cx=0012 dx=001A sp=1A48 bp=1A76 si=0082 di=1AA0
```

In the register dump, ES:BX is 0000:046CH. The CPU generated a GP fault when we tried to dereference this pointer. DOS/16M's Int 0DH handler caught the fault and shut down the application, just as OS/2 would do (OS/2 displays a similar register dump when an application GP faults). Note that even in the event of extremely serious bugs, the program does not crash or hang the machine; it exits cleanly back to the operating system.

Protected mode forbids you to peek or poke any memory not owned by your task. So how do you examine well-known memory locations on the PC? We saw earlier that DOS/16M provides a large number of "transparent" selectors where, for example, protected-mode selector B800H corresponds to real-mode segment B800H. These transparent selectors are mapped into your task's address space, so you can freely peek or poke their corresponding absolute physical addresses.

Why, then, didn't peeking at 46CH work? Apparently, selector 0 is not a transparent selector. Under Instant-D, the debugger displays this message:

```
Interrupt 0D is a general protection exception.
Null selector.
```

In protected mode, any selector < 8 references descriptor zero in either the GDT or LDT, and is considered the null selector. Any number less than eight (usually zero) can be loaded into a segment register, but attempting to dereference the resulting pointer causes a GP fault. This way, protected mode provides support for the high-level language concept of a *null pointer*. But this also means that selector 0 *can't* be used as a transparent selector to the 64K starting at memory location zero.

This is not such a big loss. There is no point in directly reading the low-memory real-mode interrupt vector table in protected mode. And the next block of memory, the BIOS data area beginning at paragraph 40H, *is* transparently handled by DOS/16M selector 40H.

By using "pointer normalization" on the real-mode address (which is illegal for a protected-mode address), we can access the tick count through selector 40H instead of illegally trying to dereference selector 0:

```
#define TICKS()   *((unsigned long far *) 0x40006CL) // 0040:006C
#define SECONDS() (TICKS() / 18)
// ...
unsigned long t1 = SECONDS();
```

Now it works. And this code works just as well in real mode, so it doesn't require #ifdef DOS16M.

The same principle applies in assembler. For example, to check if any key has been pressed, on a PC you might say:

```
xor bx, bx
mov es, bx
mov ax, es:[41Ah]
cmp ax, es:[41Ch]
```

But in order to run under DOS/16M also, this must be changed to:

```
mov bx, 40h
mov es, bx
mov ax, es:[1Ah]
cmp ax, es:[1Ch]
```

While this change isn't necessarily portable to other DOS extenders (Eclipse's OS/286, for instance, does not provide selector `40H` as a built-in transparent selector), all DOS extenders *do* provide some mechanism for creating a protected-mode selector and setting its physical base address. While the resulting selector is not transparent, it does provide access to a location in memory. In DOS/16M, you would use the functions `D16ProtectedPtr()` or `D16SegAbsolute()`, and in OS/286 you would use the "extended segment service" `Int 21H AX=E803H` ("create real window"), which we will use when we talk more about OS/286. In the above example, the selector returned from this service would then be used rather than the constant `40H`.

In keeping with known rules of software engineering, you will probably spend 95 percent of your time porting a few lines of code to protected mode, while the vast majority of the code will take no time at all.

Bugs!

One other item must be added to the list of things that could cause a DOS program to GP fault in protected mode: code that ought never to have worked in the real-mode version of your program, but somehow did.

For example, if you run the following tiny program in real mode and forget the command-line argument, it prints out a zero:

```
/* BAD.C */
#include <stdlib.h>
#include <stdio.h>
main(int argc, char *argv[])
{
    printf("%d\n", atoi(argv[1]));
}
```

```
C:\BOOK>bad 100
100
C:\BOOK>bad
0
```

But in protected mode, forgetting the command-line argument causes a protection violation:

```
C:\BOOK>bad
DOS/16M Protected Mode Run-Time     Version 3.73
Copyright (C) 1987,1988,1989 by Rational Systems,Inc.

DOS/16M: Unexpected Interrupt=000D  at 0080:12EC
code=0000 ss=00A0 ds=0000 es=00A0
ax=0000 bx=0000 cx=0012 dx=0000 sp=17A0 bp=17AC si=0000 di=17
```

The protected-mode response is actually the correct one. In real mode, the program works by accident. There is almost a magical quality to real mode: practically nothing is illegal, so almost any operation does *something* and consequently many things work even though they ought not to. In protected mode, fewer operations are legal. This is a case where limitations and restrictions actually help by hindering.

DOS/16M or OS/2 versions of a program often flush out problems. Whereas the behavior of a buggy real-mode program often depends on the current contents of uninitialized memory, a protected-mode program usually does not exhibit such seemingly nondeterministic behavior.

#ifdef DOS16M: The DOS/16M Library

The DOS/16M library provides a small but powerful set of functions for memory management and interrupt handling in protected mode, and is provided in source form (DOS16LIB.C and DOS16.H) along with DOS/16M and Instant-C. Functions that make up the library are shown below:

Access Protected-Mode Selectors

D16AbsAddress	Return absolute address of protected-mode pointer
D16GetAccess	Return access-rights byte of protected-mode pointer
D16SegLimit	Return limit of protected-mode pointer

Allocate Protected-Mode Selectors

D16ProtectedPtr	Create protected-mode pointer from real-mode pointer
D16RealPtr	Create real-mode pointer from protected-mode pointer

D16SegAbsolute	Create protected-mode selector for absolute address
D16SegCSAlias	Create executable alias, e.g., for data segment
D16SegDSAlias	Create read/write data alias, e.g., for code segment
D16SegDataPtr	Create data selector, base is offset of another pointer
D16SegTransparent	Create transparent selector to real-mode segment
D16MemAlloc	Allocate a data segment
D16HugeAlloc	Huge-model allocator; can allocate blocks > 1 megabyte

Alter Protected-Mode Selectors

D16SegProtect	Set segment to be read-only or read/write
D16SegRealloc	Set segment allocation to current strategy
D16SegResize	Set segment size
D16HugeResize	Huge-model reallocator
D16SetAccess	Set selector access-rights byte
D16SegCancel	Cancel a protected-mode selector
D16MemFree	Free a protected-mode segment and cancel its selector
D16MemStrategy	Set memory-allocation strategy, e.g., MTransparent

Query Functions

D16ExtAvail	Return bytes of extended memory available
D16LowAvail	Return bytes of DOS-managed memory available
D16isDOS16M	Return true if running under DOS/16M
_is_pm	Return true if CPU in protected mode

Interrupt Handling

D16IntTransparent	Install protected-mode BIOS interrupt handlers
D16Passdown	Set protected-mode interrupt to resignal in real mode
D16Passup	Set real-mode interrupt to resignal in protected mode
D16pmGetVector	Get protected-mode interrupt vector
D16pmInstall	Set protected-mode interrupt vector
D16rmGetVector	Get real-mode interrupt vector
D16rmInstall	Set real-mode interrupt vector
_intflag	Set CPU interrupt-enabled flag

Calling Real-mode Code

D16rmInterrupt	Real-mode interrupt, set real-mode segment registers
D16rmRCall	Real-mode far call, set real-mode segment registers

Miscellaneous

D16MoveStack	Switch location of the CPU stack
D16ToReal	Switch CPU to real mode
D16ToProtected	Switch CPU to protected mode
D16SelReserve	Reserve selectors to be used for D16HugeResize
D16TermFunction	Install callback function for program termination
D16CallDebug	Programmatic interface to Instant-D debugger

Some of these functions are similar to functionality provided at different levels in OS/2. `D16SegCSAlias`, which creates an executable (CS) alias selector for a data segment, is similar to OS/2's `DosCreateCSAlias`. `D16SegAbsolute`, which can map any absolute physical address into the user's address space, is like a more powerful version of OS/2's `VioGetPhysBuf` (which is limited to addresses from `A0000H` to `BFFFFH`). Whereas only OS/2 device drivers are allowed total access to the entire range of absolute addresses (via the `PhysToUVirt DevHlp`), under DOS/16M, any "normal" application can call `D16SegAbsolute`.

The DOS/16M interrupt-handling functions are most useful when you write real-mode ISRs. In a DOS extender, `Int 21H AH=25H` and `AH=35H` work with protected-mode ISRs, so DOS code that installs, for example, a divide-by-zero interrupt handler or a critical error handler, does not have to be changed. Furthermore, DOS/16M classifies most real-mode interrupts as "auto-passup," so they are automatically resignaled in protected mode. To establish real-mode interrupt handlers that do *not* cause a mode switch (this is what you would need in the case mentioned earlier of 9600 baud asynch communications on an IBM AT), you need `D16rmInstall()` and `D16rmGetVector()`. The functions `D16pm-Install()` and `D16pmGetVector()` are in most cases identical to protected-mode `Int 21H AH=25H` and `AH=35H`, except that the DOS/16M functions are useful when installing interrupt handlers for processor exceptions such as `Int 0DH`, the GP fault (see "Stalking GP Faults," Part 1, *Dr. Dobb's Journal*, January 1990).

Many of the DOS/16M functions allocate selectors in the GDT. For example, `D16SegAbsolute` does not allocate any memory, but it must use a selector. DOS/16M sets the selector's base address to point to an absolute memory location. Likewise, `D16SegCSAlias` must allocate a selector: DOS/16M copies a data segment's selector into the new selector, and then changes the new selector's access-rights byte so that it is executable.

While protected mode on a 286 offers a large address space, the descriptor tables are themselves just data segments and can therefore be no larger than 64K in

16-bit protected mode. Consequently, selectors are a limited resource. You must therefore be prepared for calls like `D16SegCSAlias` and `D16SegAbsolute` to fail.

Some of these functions are used in the following program, NB.C, which tests whether NetBIOS (a semi-portable network communications protocol) is loaded. NetBIOS (`Int 5CH`) is a typical real-mode service for which DOS extenders do not provide transparent access. The function `netbios_loaded()` places an invalid command in a NetBIOS control block (NCB), puts the address of the NCB in `ES:BX`, and generates `Int 5CH`. On return from `Int 5CH`, if the retcode field in the NCB is set to `ERROR_INVALID_COMMAND`, NetBIOS is present.

Since NetBIOS runs in real mode, the address expected in ES:BX must be a real-mode address. The protected-mode version of NB.EXE can't simply allocate an NCB and put its address in ES:BX. Instead, the NCB must be allocated *in low memory* and, when `Int 5CH` is resignaled in real-mode, ES must contain a real-mode segment.

Under DOS/16M, NB.C tries to allocate a "transparent" selector and, failing that, tries to allocate a low-memory selector. Fields of the NCB are set and tested using the protected-mode pointer, but its real-mode equivalent is passed to NetBIOS. Since the `Int` instruction in protected mode invokes a protected-mode interrupt, and since DOS/16M installs protected-mode handlers for only the most important interrupts, NB.C uses `D16rmInterrupt()`, which invokes a real-mode interrupt, and sets the real-mode segment registers:

```
/*
NB.C -- test for presence of NetBIOS

    MSC 5.1:
        cl -AL -Ox nb.c

    DOS/16M:
        cl -AL -DDOS16M -c -Zi nb.c
        link preload crt0_16m pml nb dos16lib /map/noe/co,nb;
        makepm nb
        splice nb nb

    DOS/16M with NETBIOS package:
        ; with packages, don't compile with -DDOS16M
        cl -AL -c -Zi nb.c
        link \16m\preload \16m\crt0_16m \16m\pml nb /map/noe/co,nb;
        makepm nb
        splice nb.exe \16m\packages\net5c.exp nb.exp \16m\loader.exe

    Instant-C:
        _CodeView = 1;
        _struct_alignment = 1;
        #defineg DOS16M
        #load dos16lib.c
        #load nb.c
        #run
*/

#if ! defined(M_I86CM) && ! defined(MI_86LM)
#error "Requires large data pointers"
#endif

#pragma pack(1)

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#ifdef DOS16M
#include "dos16.h"
#endif

typedef enum { preferext, preferlow, extended, low, transparent
    } STRATEGY;
typedef enum { FALSE, TRUE } BOOL;
typedef unsigned char BYTE;
typedef char far *FP;
typedef unsigned WORD;
typedef unsigned long DWORD;

#define MK_FP(seg,ofs)  ((FP)(((DWORD)(seg) << 16) | (ofs)))

typedef struct {
    BYTE command, retcode, lsn, num;
    FP buffer;
    WORD len;
    BYTE callname[16], name[16];
    BYTE rto, sto;
    FP postrtn;
    BYTE lananum, cmdcplt;
    BYTE reserved[14];
    } NCB;

#define NETBIOS_INT                     0x5C
#define INVALID_COMMAND                 0x7F
#define ERROR_INVALID_COMMAND           0x03

BOOL netbios_loaded(void);
void netbios_request(NCB far *ncb);
```

```
FP alloc(WORD size, STRATEGY strat);
void dealloc(FP fp);
FP getvect(WORD intno);

BOOL trans = TRUE;

void fail(char *s) { puts(s); exit(1); }

main()
{
    BOOL net = netbios_loaded();
    puts(net ? "NetBIOS loaded" : "NetBIOS not loaded");
    return !net;    // MS-DOS ERRORLEVEL 0 = success
}

BOOL netbios_loaded(void)
{
    if (! getvect(NETBIOS_INT))
        return FALSE;
    else
    {
        BOOL ret;
        NCB far *ncb = (NCB far *) alloc(sizeof(NCB), transparent);
        if (ncb == NULL)
        {
            if ((ncb = (NCB far *) alloc(sizeof(NCB), low)) == NULL)
                fail("cannot allocate low-memory pointer");
            else
                trans = FALSE;
        }
        ncb->command = INVALID_COMMAND;
        netbios_request(ncb);
        ret = (ncb->retcode == ERROR_INVALID_COMMAND);
        dealloc((FP) ncb);
        return ret;
    }
}

void netbios_request(NCB far *ncb)
{
#ifdef DOS16M
    NCB far *real_ncb = (trans) ? ncb : D16RealPtr(ncb);
    D16REGS r;
    r.es = FP_SEG(real_ncb);
    r.bx = FP_OFF(real_ncb);
    /* signal interrupt in real mode, set real-mode segment regs */
    D16rmInterrupt(NETBIOS_INT, &r, &r);
#else
```

```
    union REGS r;
    struct SREGS s;
    s.es = FP_SEG(ncb);
    r.x.bx = FP_OFF(ncb);
    int86x(NETBIOS_INT, &r, &r, &s);
#endif
}

/* INT 21H AH=35H in pmode returns pmode interrupt vector */
/* to get real-mode vector, call D16rmGetVector */
FP getvect(WORD intno)
{
#ifdef DOS16M
    INTVECT iv;
    return D16rmGetVector(intno, &iv) ? MK_FP(iv.sel, iv.off) : (FP) 0;
#else
    return _dos_getvect(intno);
#endif
}

FP alloc(WORD size, STRATEGY strat)
{
#ifdef DOS16M
    STRATEGY old = D16MemStrategy(strat);  /* set allocation strategy */
    FP fp = D16MemAlloc(size);             /* can't use calloc here! */
    /* can't use calloc, but must zero out structure */
    /* could also use MAKEPM -INIT00 option */
    if (fp) memset(fp, 0, sizeof(NCB));    /* large-model memset */
    D16MemStrategy(old);                   /* restore previous strategy */
    return fp;
#else
    return calloc(1, size);
#endif
}

#ifdef DOS16M
void dealloc(FP fp)     { D16MemFree(fp); }
#else
void dealloc(FP fp)     { free(fp); }
#endif
```

There is no single DOS/16M call to allocate out of low memory. Instead, you first call `D16MemStrategy()` to set the DOS/16M memory-allocation strategy, then allocate memory, and then restore the previous allocation strategy. To move a block of memory from, for example, extended to low memory, call `D16Mem-Strategy()`, then call `D16SegRealloc()`, which compares the segment's allocation with the current strategy. The DOS/16M allocation strategies are prefer-extended, prefer-low, force-extended, force-low, transparent, and transparent-stack.

Note that after setting the strategy, NB.C does not call C memory management routines such as `malloc()` or `calloc()`. These functions allocate out of pools of storage, and are not affected by the DOS/16M allocation strategy until the next time they happen to call DOS to add to their pools. To guarantee that allocation requests reflect the current strategy, `D16MemAlloc()` is called instead.

While NB.C manages to isolate the differences between real and protected modes inside subroutines like `netbios_request()` and `alloc()`, clearly this code is nonetheless heavily dependent on DOS/16M.

This same code can run under DOS/16M, without explicitly using the DOS/16M library, and without any `#ifdef DOS16M` code. All we do is splice in the NET5C package supplied with DOS/16M:

```
        splice nb.exe packages\net5c.exp nb.exp
```

While packages provide an elegant way to completely isolate changes for protected mode, packages themselves must be written using the DOS/16M library, so they don't eliminate the need to know about the DOS/16M functions. Behind the scenes, the NET5C.EXP package uses `D16pmInstall()` to install a protected-mode interrupt handler for `Int 5CH`, and to install a handler for NetBIOS "post" (asynchronous callback) routines. The `Int 5CH` handler itself uses such DOS/16M routines as `D16RealPtr()` and `D16rmInterrupt()`. The result is that `Int 5CH` is handled almost as transparently as is `Int 21H`.

In addition to using the NET5C package, the header to NB.C indicates yet another way to run the program in protected mode: using the Instant-C development environment. All the DOS/16M functions can be called from within Instant-C, and, as we will see later on, Instant-C provides an excellent environment in which to work with the DOS/16M library and explore protected-mode C programming in general.

OS/286 and the Two-Machine Model

It is important that we look at another DOS extender: OS/286, from Eclipse Computer Solutions, formerly A. I. Architects. Most code ported to DOS/16M also runs under OS/286, but there are a number of important differences between the two products.

Whereas Rational Systems' first product was Instant-C, and DOS/16M was an outgrowth of this work, A. I. Architects started out as a hardware company, making the HummingBoard, a 386 protected-mode coprocessor for PCs. The best-known client for the HummingBoard is Gold Hill, makers of LISP systems for the PC. Unlike other 386 accelerator boards that replace the PC's CPU, the HummingBoard is used *in addition to* the native CPU, much like a math or graphics coprocessor.

This experience with running a protected-mode processor alongside an existing real-mode processor is the basis for Eclipse's DOS extender products. Even when OS/286 runs on a single processor, switching it between real and protected mode, the OS/286 model is: *DOS extender as coprocessor.*

One of the OS/286 architects, Fred Hewett, explained this in a paper, "DOS Extenders and the Two-Machine Model":

> It is useful to think of the 286 (or 386) as two distinct microprocessors sharing a single package. The core of a DOS extender is a communication system between the two machines.... Interestingly, in the initial implementation of OS/x86, the real-mode processor and the protected-mode processor were physically different chips. The real-mode chip was the 80x86 of a PC, and the protected-mode processor was a 386 on an add-in card. The architecture of OS/x86, which began with a true two-processor mode, reflects the dual nature of the 286 and 386 processors.

In this model, the real-mode machine is used for I/O and user interface, while the protected-mode machine is used for memory and task management.

The two machines communicate via "real procedure calls" (RPC), a name intentionally similar to the "remote procedure calls" used in networking. The machines do not share address space, even when inhabiting the same processor: we've seen several times that protected-mode selectors have no meaning in real mode, and that real-mode segment values have no meaning in protected mode. (Transparent or bimodal selectors only *look* like an exception to this rule.) Since the real- and protected-mode address spaces are disjoint, the two modes can be viewed as a *very* local area network. Pointers cannot be shared between the two modes, any more than they can be shared between two nodes on a network; communication must be by value, not by reference. Thus, RPC is an excellent model for the address translation performed by a DOS extender.

But aside from having a perhaps more thought-out philosophical outlook on DOS extenders than does DOS/16M, how does the two-machine model of OS/286 relate to actual differences between the two products?

In some ways, of course, it doesn't. When implemented on a single processor, for example, OS/286's RPC is no different from DOS/16M's interrupt chaining with a mode switch. Interrupts are used for inter-mode communication.

At least one crucial respect of OS/286 differs substantially from DOS/16M: OS/286 maintains a rigid separation between your protected-mode application on the one hand, and the OS/286 kernel on the other. We've already seen that DOS/16M runs your application at the same protection level as the kernel (Ring 0), and that DOS/16M uses the GDT, rather than an LDT, for your application's selectors. In contrast, the OS/286 kernel uses the GDT and runs in Ring 0, while your task has its own LDT and runs in Ring 3.

In a multitasking operating system such as OS/2, the distinction between the kernel on the one hand and applications on the other is crucial. But what about in the world of DOS extenders, where only one task is running, and where the kernel exists largely in order to run this one task? Is the distinction between kernel and task still useful?

Several trade-offs are inherent in OS/286's use of Ring 3 and the LDT. Because of the decision to use Ring 3 for applications and Ring 0 for the kernel, some operations involved with interrupt handling become more difficult or more time-consuming for DOS applications ported to protected mode. In an informal test, a program generated 200,000 software interrupts in 23 seconds in real mode, 32 seconds under DOS/16M, and 51 seconds under OS/286.

OS/286 is much closer to the architecture of a genuine protected-mode operating system such as OS/2, while DOS/16M is much closer to DOS. (Even the names reflect this.) While porting to OS/286 probably involves more work than porting to DOS/16M, you might find that it puts you closer to the goal of OS/2 compatibility (if that is, in fact, your goal). While not an OS, OS/286 has more of a bona fide kernel than DOS/16M.

Still, this helps explain why OS/286 is generally less convenient to work with than DOS/16M, and why it is larger and a bit slower: OS/286 has "more architecture" than DOS/16M. There are also all sorts of issues involving privilege transitions, gates, and the protected-mode TSS, all of which are important, but none that we need to discuss here.

Another aspect of OS/286's clear demarcation between kernel and application is that the OS/286 kernel is loaded as a separate TSR. OS/286 is a little more akin to an environment than is DOS/16M. Before running OS/286 executables, one first runs the OS286.EXE TSR. This can be bound into OS/286 executables, but they then become extremely large. Once the kernel has been installed, executables can be run using a loader called UP.EXE, or by running a bound executable which includes UP.EXE, or by using a custom loader. The OS/286 manual devotes an entire chapter to writing your own loader, which can communicate with the OS/286 kernel. Finally, OS/286 comes with a combined command processor and debugger, CP, which runs under the kernel, and which can load and run both real-mode and protected-mode executables.

This brings up one interesting benefit to OS/286's architecture. Because OS/286 uses the LDT, the DOS EXEC function in OS/286 (`Int 21H AH=4BH`) is able to spawn protected-mode .EXP files: the kernel owns the GDT, each task gets its own LDT, and parent and child tasks can share selectors in the GDT. In contrast, DOS/16M, which uses the GDT for everything, has less opportunity for handling more than one protected-mode task.

OS/286 does not support full multitasking, however. Instead, it just supports "task management" à la MS-DOS, where‹a parent can spawn a child task but must wait for it to complete.

Of course, DOS also has an unofficial form of multitasking: the TSR. DOS/16M supports `Int 21H AH=31H` ("terminate and stay resident") in protected mode, allowing you to write protected-mode TSRs. While every DOS/16M task gets its own loader and GDT, there is also limited communication between DOS/16M tasks using `Int 15H`. OS/286 also allows you to create protected-mode TSRs, but `Int 21H AH=31H` can only be called from a custom version of the loader. Again we see various trade-offs between the two DOS extender architectures.

Another important difference is that whereas Rational Systems currently only produces a 286-based product, Eclipse produces not only OS/286, but also OS/386 and OS/386 HB (HummingBoard)—see Chapter 5 for more information. The compatibility between these products is very high. All share the same manual. The OS/286 API is directly based on OS/386, and in fact the functions needed to write your own loader bear such names as `OS386_Get_GDT()` and `OS386_Create_Task()`, even when used in OS/286. The Eclipse extensions to DOS were designed around 32-bit values. While each Eclipse product has its own kernel (OS286.EXE versus OS386.EXE), the tools that communicate with each kernel are identical. Thus, EXPRESS.EXE, CP.EXE, and UP.EXE in OS/286 are identical to the versions included with OS/386. This is a major advantage to keeping the kernel separate from utilities that talk directly to your application.

At this point, we had better discuss these utilities. Just as we did with DOS/16M, to take the LIST program and prepare it for OS/286, we can start with the same LIST.OBJ as in real mode. The next steps are:

```
 link /noe/map list,list,list,\os286\llibce;
\os286\express list
\os286\bind -o list.exe -l \os286\tinyup.exe \
   -k \os286\os286.exe -i list.exp
```

Whereas DOS/16M provides loose .OBJ modules for you to link in with your application, OS/286, during installation, makes a copy of your real-mode library and adds in its .OBJ modules. To link, you simply specify this .LIB rather than use the one that came with the compiler.

An important point is that the resulting executable *will still run in real mode.* The OS/286 .OBJ modules simply correct flaws in the real-mode library, but don't otherwise tie it to protected mode or to OS/286. In contrast, some of the DOS/16M .OBJ modules prevent the executable from running in real mode, because the LINK output is just meant as preparation for MAKEPM.

At this stage, whereas for DOS/16M you run the MAKEPM utility, for OS/286 you run EXPRESS. EXPRESS takes a well-behaved DOS application and its .MAP file and, like MAKEPM, translates segments to selectors, producing an .EXP file. So the sole job of the OS/286 replacement .OBJ modules is to make a DOS executable well behaved. Some compiler libraries (for example, MetaWare High C) don't require patching; Microsoft C appears to be the most ill-behaved, as it requires the most patches.

While OS/286, DOS/16M, and 32-bit DOS extenders all produce protected-mode executables with an .EXP extension, this extension is about all the files have in common. There is unfortunately no common protected-mode executable file format.

The .EXP file produced by EXPRESS can, as stated earlier, be run in several ways. The OS/286 equivalent of SPLICE is the BIND utility, which must be purchased separately from Eclipse.

The OS/286 API has two levels. One, already mentioned, allows you to write custom loaders by communicating with the Eclipse kernel. As noted earlier, while bearing the prefix "OS386_," these are also used in OS/286, and include:

Initializing and Loading

OS386_Init_Machine	Boot protected mode
OS386_Create_Task	Load a task

Real Procedures and Signals

OS386_Declare_RPC	Assign ASCIIZ name to real-mode procedure

OS386_Delete_RPC	Remove a real-mode procedure
OS386_Generate_Signal	Call real-mode procedure from protected mode

Information Services

OS386_Get_Exit_Code	Get exit code
OS386_Get_Protected_Machine	Get processor type
OS386_Get_GDT	Get Global Descriptor Table selector
OS386_Get_LDT	Get Local Descriptor Table selector
OS386_Get_Segment_Info	Look at descriptor
OS386_Get_Task_ID	Get a task ID
OS386_Get_Version	Get OS/x86 version

Debugging Calls

OS386_Read_Mem	Read protected memory
OS386_Write_Mem	Write protected memory
OS386_Step_Task	Step task
OS386_Task_Control	Step, suspend, start, or kill task

The other level has roughly the same functionality as the DOS/16M library, though it uses the `Int 21H` interface rather than the C language far-call interface used by the DOS/16M library. The Eclipse extensions to `Int 21H` include:

Real Procedure Calls

AH=E0H	Initialize real procedure
AH=E1H	Issue real procedure call

Interrupts, Heap Management, and Signals

AH=E2H	Set real procedure signal handler
AH=E3H	Issue real interrupt
AH=E4H AL=00H	Chain to real-mode handler
AH=E4H AL=02H	Set protected-mode task gate
AH=E4H AL=03H	Remove protected-mode task gate
AH=E5H AL=00H	Heap management strategy
AH=E5H AL=01H	Force heap compaction
AH=E6H	Issue real procedure signal from protected mode

Extended Segment Services

AH=E7H	Create code segment
AH=E8H AL=00H	Create data segment

AH=E8H AL=01H	Create data window/alias
AH=E8H AL=02H	Create real segment
AH=E8H AL=03H	Create real window/alias
AH=E8H AH=06H	Create shareable segment
AH=E9H AL=01H,02H	Change segment parameters (code/data)
AH=E9H AL=05H	Change segment parameters (adjust limit)
AH=E9H AL=06H	Change segment parameters (base address)
AH=EAH	Allocate multiple windows (huge segments)
AH=ECH	Block transfer
AH=EDH	Get segment or window descriptor

Some of these calls are used in the following modifications to the program NB.C, which allocated a low-memory protected-mode selector (possibly a transparent one) and issued a real-mode `Int 5CH`, to test if NetBIOS is present. The upper half of the program is unchanged; only the alterations for OS/286 are shown here:

```
#ifdef OS286
/* given protected-mode pointer, returns physical base address */
DWORD prot2abs(FP fp)
{
    union REGS r;
    r.h.ah = 0xED;  /* "get segment or window information" */
    r.h.al = 0x02;  /* real segment */
    r.x.bx = FP_SEG(fp);
    intdos(&r, &r);
    return (r.x.cflag) ? 0L : (DWORD) MK_FP(r.x.si, r.x.bx);
}
#endif

void netbios_request(NCB far *ncb)
{
#ifdef OS286
    typedef struct { WORD ax,bx,cx,dx,flags,si,di,ds,es; } MACHINE_STATE;
    MACHINE_STATE state, *pstate = &state;
    union REGS r;
    struct SREGS s;
    DWORD absaddr;
    segread(&s);
    r.h.ah = 0xE3;              /* "issue real interrupt" */
    r.h.al = NETBIOS_INT;       /* interrupt number */
    r.x.dx = FP_OFF(pstate);
    s.ds = FP_SEG(pstate);
    r.x.bx = 0;                 /* don't need any return registers */
    /* now set up registers for real-mode interrupt */
    absaddr = prot2abs(ncb);
    state.es = absaddr >> 4;
    state.bx = absaddr & 0x0F;
    intdosx(&r, &r, &s);
#endif
}

FP getvect(WORD intno)
{
    /* okay for OS/286: returns address of protected-mode surrogate */
    return _dos_getvect(intno);
}

FP alloc(WORD size, STRATEGY strat)
{
#ifdef OS286
    union REGS r;
    r.x.cx = 0;
    r.x.dx = size;
    /* use "extended segment service":
       either "create real segment" or "create data segment" */
    if (strat == low) r.x.ax = 0xE802;
    else if (strat == extended) r.x.ax = 0xE800;
    else return (FP) 0;     /* not supported */
    intdos(&r, &r);
    return (r.x.cflag) ? (FP) 0 : MK_FP(r.x.ax, 0);
#endif
}

#ifdef OS286
/* use standard DOS call, even for segments allocated with
   "extended segment services" */
void dealloc(FP fp)     { _dos_freemem(FP_SEG(fp)); }
#endif
```

As with DOS/16M, we test for the presence of NetBIOS by putting an invalid command in a NetBIOS control block (NCB). This control block must be in low memory, because NetBIOS runs in real mode. The function `alloc()` uses an OS/286 "extended segment service" to allocate a segment whose physical base address is in low memory. After setting up the low-memory NCB using the protected-mode pointer, the function `netbios_request()` is called to put the real-mode address of the NCB in ES:BX, switch to real mode, issue an `Int 5CH`, and switch back to protected mode. To do this in DOS/16M, we called `D16rm-Interrupt()`. In OS/286, we use the "issue real interrupt" service. This expects the address of a `MACHINE_STATE` block in DS:DX. The fields of the `MACHINE_STATE` block hold register values destined for real mode. To get the real-mode address of the NCB to place in the ES:BX fields of the MACHINE_STATE block, we call the function prot2abs() which, in turn, uses the OS/286 "get segment or window information" service. Finally, we deallocate the low-memory segment using the Microsoft C _dos_freemem() call, which performs an Int 21H AH=49H.

Once this is substituted for the #ifdef DOS16M sections of NB.C, the program can be prepared for OS/286 with the following commands:

```
cl -AL -Ox -G2 -c -DOS286 nb.c
link /map nb,nb,nb,\os286\llibce.lib;
express nb
```

Lattice C and "Extended Family Mode"

It is also important to mention OS/286's forthcoming inclusion in the Lattice 80286 C Development System. OS/286 is already bundled with a number of other languages, including Gold Hill Lisp and Lahey Fortran.

Lattice will be using the DOS extender as part of its "Extended Family Mode" for portability between MS-DOS and OS/2. Microsoft has dubbed a small subset of the OS/2 API the "Family API": OS/2 API functions that can be called under either OS/2 or MS-DOS. Lattice is using the larger address space available under the OS/286 DOS extender to extend the range of OS/2 functionality that can also be used under MS-DOS.

The output of this process is not a .EXP file. Since family-mode applications already include a real-mode stub loader, which loads a protected-mode OS/2 executable in DOS and connects it to real-mode API simulator functions, this same mechanism can be extended to run an OS/2 executable in protected-mode MS-DOS. Lattice's LBIND utility, which attaches a stub loader to an OS/2 executable, can be used to attach a DOS extender loader (similar to OS/286's UP.EXE) to an "Extended Family Mode" application.

Lattice president David Schmitt argues that this is the future for 286 machines: "The 80286 is just too weak for OS/2. On the other hand, the 80286 is too strong for DOS."

Performance

There are two reasons for concern about the performance of programs that run under a 286-based protected-mode DOS extender. First, precisely because it is protected, protected mode is, *at a raw level*, inherently slower than real mode. We saw earlier that the LES BX instruction requires more clocks in protected than in real mode. As Table 4-2 shows, the same is true for *any* 286 instruction that loads segment registers. This penalty is especially severe for large models.

Table 4-2: 286 performance.

Instruction	Real	Protected
CALLF	13+	26+m
INT	23+	(40,78)+m ; loads CS
IRET	17+	(31,55)+m
JMP FAR	11+m	23+
LES etc.	7	21
MOV seg	5	19
POP seg	5	20
RETF	15+m	25+m,55

The second reason to worry about protected-mode performance is the notoriously expensive protected-to-real mode switch on the 286. Since a DOS extender very frequently switches into real mode (not only to service explicit DOS and BIOS requests from your program, but also to handle external interrupts from the clock, keyboard, network adapter, and so on), one might think such a program would be unusable on a 286.

In fact, the protected-mode version of a large piece of commercial software generally performs *better* than the real-mode version. The reason is not difficult to find. Contrary to the often noted time/space trade-off in software, in many programs that are cramped for space, *there is no such trade-off*: giving the program more space makes it faster. This is particularly true for programs that use overlays or some form of virtual memory (VM). When a program doesn't have to spend all its time inside the VM manager (which is pure overhead), it is able to get some actual work accomplished. A program that doesn't have a time/space trade-off usually performs better in protected mode than in real mode.

On machines with souped up "power user" CONFIG.SYS files, the real-mode version may still perform better than the DOS extender version. Interestingly, though, the performance of the DOS extender version is largely insensitive to any installed speedup and caching utilities. Thus, rather than advise users to install various utilities to improve your program's performance, you can provide the possibility of simplified configuration by using a DOS extender.

In one piece of commercial software with its own built-in VM, the vastly increased headroom brought about by DOS/16M did introduce one major problem: while the VM "garbage collector" was not getting called anywhere as frequently as under real mode, *when it did eventually get called,* it had enormous amounts of garbage to mark and sweep, and would take forever. To cope with the larger memory available in protected mode, the garbage collector had to be rewritten to operate incrementally.

On a related issue, one also needs to be concerned about *badly behaved heaps.* The memory-management routines in PC compiler run-time libraries such as MSC 5.1 were not written to handle megabytes of memory. Microsoft's startup code, in fact, establishes an upper bound of 20 heaps. The DOS/16M and OS/286 startup codes change this to a larger number, but the heap routines themselves do not expect to be handling huge lists.

The LIST.C program presented at the beginning of this chapter defaults to creating 512-byte nodes, but if we tell it to create much smaller nodes, the program performs more calls to `malloc()`. When the list gets very large, the performance of the heap code totally falls apart. Allocating six-byte nodes, it takes the real-mode version only one second to allocate the first 22,000 nodes, but allocating the last 1,000 nodes takes 185 seconds. This seems pretty badly behaved, but the disparity between the early allocations and the later allocations is even worse in protected mode: whereas the first 75,000 nodes can be allocated in 72 seconds, allocating the last 1,000 nodes (6K) takes *ten minutes!*

When memory is finally exhausted, the protected-mode version has allocated 80,000 nodes in 1,700 seconds (45 nodes/second), and the real-mode version has allocated only 25,000 nodes in 500 seconds (50 nodes/second), so the overall performance per node is about the same. But in protected mode, it's that last 10 minutes that kills you. This is definitely something to plan for.

Returning to normal behavior, Table 4-3 shows run-time and allocation figures for the LIST program, using 512-byte nodes, and running on three different machines:

Table 4-3: Run-time and allocation figures for LIST.

	DOS/16M			*Real Mode*		
Machine	alloc	sec	k/sec	alloc	sec	k/sec
IBM PC/AT 8 MHz (2 meg)	1714K	22	77	454K	5	90
Compaq 386/20e (4 meg)	3240K	16	202	500K	2	250
PS/2 Model 80 (6 meg)	5575K	31	179	505K	2	253

Mainly, this shows that the real-mode version is, as expected, oblivious to the amount of memory installed on a machine. On the PS/2 Model 80 with 6 megabytes of memory, the DOS/16M version allocates more than 10 times as much memory as the real-mode version. And remember, the source code for the two versions is identical. But these throughput figures do also show that this DOS/16M version consistently allocates fewer kilobytes per second than the real-mode version.

While this toy program is hardly representative of the large applications that are likely to use a DOS extender, and while large protected-mode applications are likely to be *faster* than their real-mode version, it is still useful to look at the LIST program and ask, where is it spending its time?

The DOS/16M debugger *sampler* command controls a built-in performance monitor. In addition to showing the amount of time spent in each function, the performance monitor also reveals time spent in protected versus real mode, as well as the number of mode switches and interrupts, and provides a complete census of DOS calls.

On the IBM AT, the LIST program spends about 3/4 of its time in protected mode and 1/4 in real mode. The program does about 3,300 mode switches, almost all of them to service `Int 21H` requests. The most frequently-called DOS request is `Int 21H AH=4AH (realloc)`, which accounts for over seven seconds of run-time! Microsoft's internal routines `amalloc()` and `mem0()` account for over 90 percent of the time spent in user code.

In addition, the LIST program itself displays the elapsed time for each 1,000 nodes it allocates. This display, together with our discussion of badly-behaved heaps, indicates that the program spends the bulk of its time allocating the last few nodes: time seems to pass more slowly toward the end.

This suggests that one way to optimize this program is to tinker with DOS/16M's allocation strategy. For example, we can force the program to use only extended memory, either by calling `D16MemStrategy()` at the beginning of the program, or, if we want to a avoid coding in DOS/16M dependencies, by using a MAKEPM option to "force extended":

```
        makepm -mfx list
```

On the IBM AT, the DOS/16M version now allocates only 1268K, but this is still three times as much as in real mode. Meanwhile, the run time is slashed from 22 to nine seconds. Pretty good for throwing one MAKEPM switch, but the loss of almost 500K on a two-megabyte machine is probably unacceptable. We need a way to maintain this performance, while still allowing low-memory allocations.

One way is to keep the change that forces extended-memory allocations, but to switch over to low memory when extended memory is exhausted. This does require inclusion of "dos16.h" and use of the DOS/16M library, but actual pointer handling remains unchanged. The changed code inside the allocation loop now reads:

```
if ((q->data = malloc(nodesize)) == NULL)
{
    D16MemStrategy(MForceLow);
    /* try again */
    if ((q->data = malloc(nodesize)) == NULL)
    {
        /* exhausted all memory in the machine */
        free(q);
        break;
    }
    printf("Switched to low memory!\n");
}
```

The new throughput figures for the program, alongside those for the original DOS/16M and real-mode versions, are shown in Table 4-4.

Table 4-4: Throughput figures for LIST.

Machine	alloc	sec	k/sec	Original	Real Mode
IBM PC/AT 8 MHz (2 meg)	1708K	13	131	77	90
Compaq 386/20e (4 meg)	3201	12	266	202	250
PS/2 Model 80 (6 meg)	5566	30	185	179	253

On both the two-megabyte and four-megabyte computers, the DOS/16M version now not only allocates far more memory than the real-mode version, but is also faster. Clearly, this optimization plays an important role on computers where low memory is a significant percentage of total memory on the machine.

Benefits and Limitations of 286-based DOS Extenders

In conclusion, it would be useful to review the advantages and disadvantages of 286-based protected-mode DOS extenders, compared with each of: 640K MS-DOS, 386-based DOS extenders, and OS/2.

MS-DOS

The advantages of DOS/16M and OS/286 over "plain vanilla" MS-DOS are pretty obvious: access to up to 16 megabytes of memory versus access to 640K of memory, and hardware-assisted memory protection versus a total absence of rules. Because of the larger real estate available in protected mode, large applications may also run faster in protected mode than in real mode.

But 286-based DOS extenders have a number of disadvantages you should be aware of. Users must have an IBM AT, or better, to run a DOS-extended program. 8088-based PCs are definitely on the way out, but the XT is still dominant in Europe. Furthermore, for small programs, protected mode can be slower than real mode, and switching between protected and real mode on a 286 machine may be too slow for some applications.

One other disadvantage of a DOS extender is that the vastly increased resources suddenly available, with very little work, may seem like a license to write *really bad code*. Limitations aren't always such a bad thing; a lot of programs out there would benefit from having some stringent limitations placed on them.

386-based DOS Extenders

DOS/16M and OS/286 have three primary advantages over 32-bit DOS extenders like OS/386 and Phar Lap's 386 | DOS-Extender (which are discussed in detail in the following chapter). First, programs developed with DOS/16M and OS/286 can run on IBM PC/ATs or other 286-based PC compatibles, whereas 32-bit DOS extenders require 386-based or higher computers. Second, because 32-bit applications generally use a linear address space with very little segmentation, they do not have the debugging advantages of highly segmented 16-bit protected-mode programs. Third, moving from 16-bit real mode based on the 8088 to 16-bit protected mode based on the 80286, requires fewer changes to your source code than moving all the way to 32-bit protected mode.

The advantages of 32-bit protected mode over 16-bit protected mode are tremendous: anyone who has seen the code produced by 32-bit C compilers never wants to go back to 16-bit code. Make no mistake, 32-bit code is the wave of the future. Furthermore, while 286-based protected mode offers 16 megabytes of physical memory, any individual item cannot be larger than 64K, unless you resort to huge model. In contrast, 32-bit protected mode allows objects as large as four gigabytes. And while the lack of segmentation when using a single linear address space in 32-bit protected mode means you lose some debugging advantages, in general, most PC programmers would be happy if they never saw another segment.

OS/2

Finally, what about OS/2? There seem to be three reasons to use DOS/16M or OS/286 rather than OS/2. First, porting to a 286-based DOS extender may take only days or weeks, in contrast to an OS/2 port, which may take months, or even involve a total rewrite of your program. Second, with a DOS extender you do not have to convince your users to buy a new operating system: DOS/16M and OS/286 work with the MS-DOS 3.x they already have. Finally, in contrast to OS/2, 286-based DOS extenders perform reasonably on 286 machines. One ad for the Glockenspiel C++ compiler reports that in one test, the OS/2 version took 2.45 minutes as opposed to 45 seconds under the DOS extended version.

But the ease of working with DOS extenders, and their raw performance advantages, should not blind us to the benefits of OS/2. In fact, the reason it takes so much work to port to OS/2 is that the benefits are so great. In addition to the large address space and memory management offered by DOS extenders, OS/2 offers virtual memory, multitasking, inter-process communications, graphics, and windows. A program such as Lotus 1-2-3/G (the OS/2 Presentation Manager version of 1-2-3) simply could not be written without such facilities.

The ability to do several things at once largely diminishes the importance of raw throughput figures: while the DOS/16M version of a product takes only 45 seconds to do something for which the OS/2 version requires 2.45 minutes, on the other hand let's not forget that during those 45 seconds, *you can't do anything else*. It's strictly one thing at a time in the world of DOS extenders (though DOS/16M does support traditional DOS pseudo-multitasking techniques such as TSRs, and both OS/286 and DOS/16M programs can run inside the DESQview multitasker).

Of course, not every application *needs* OS/2's advanced features, and you can't always go on to something else while the compiler is taking 2.45 minutes to compile your program. Where OS/2 might be perfect for one application, a DOS extender, or perhaps even real-mode MS-DOS itself, might be right for another. Actually, all these different environments complement each other rather nicely. There is no one perfect operating system, any more than there is one perfect programming language, or one true method for brewing tea. Perhaps in the future we will see a loose merger of real-mode MS-DOS, 286- and 386-based protected-mode MS-DOS, and OS/2, with developers able to freely use whichever one is appropriate for the job at hand.

Programming Project

Exploring Protected Mode with Instant-C

Almost all the programs and code fragments in Chapter 4 can be tested using Rational Systems' Instant-C, an interactive protected-mode C compiler and integrated development environment, based on DOS/16M, for 286- and 386-based PC compatibles.

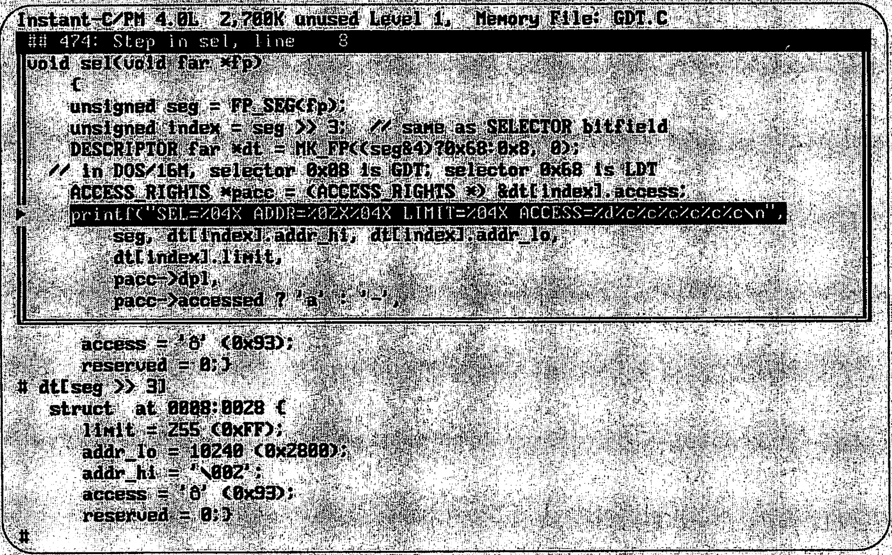
Instant-C (IC) provides interactive execution, linking, editing, and debugging of C code. In addition to loading .C files, C expressions can be typed in at IC's # prompt for immediate evaluation, at global scope, outside any function. Figure 4-5 offers a look at Instant-C.

The following example uses the Microsoft C `_dos_allocmem()` function and the DOS/16M `D16MemStrategy()` function, first to allocate extended memory, and then to allocate low memory. Note how C statements, declarations, and preprocessor statements can be freely mixed at the # prompt, somewhat like mixing statements and declarations in C++. Note also that leaving the semicolon off a statement tells IC to print its value; this is one way that interactive C differs from "normal" C:

```
# #include "dos16.h"
DOS16.H included.
# unsigned seg, seg2;
# D16MemStrategy(MForceExt);
# #include <dos.h>
DOS.H included.
# _dos_allocmem(1000, &seg);
# seg
    3080 (0xC08)
# D16AbsAddress(MK_FP(seg,0))
    1910176 (0x1D25A0)
# D16MemStrategy(MForceLow);
# _dos_allocmem(1000, &seg2);
# seg2
    3088 (0xC10)
# D16AbsAddress(MK_FP(seg2,0))
    236064 (0x39A20)
```

IC can be used as a test bed in which functions are tried out, variables are declared, expressions are evaluated, and so on, without actually writing a program. At the same time, the product manages to maintain fairly high compatibility with ANSI C, including full support for function prototypes.

Figure 4-5: Instant-C, a protected-mode C development environment.



```
Instant-C/PM 4.0L  2,700K unused Level 1,  Memory File: GDT.C
 ## 474: Step in sel, line    8
void sel(void far *fp)
    {
    unsigned seg = FP_SEG(fp);
    unsigned index = seg >> 3;  // same as SELECTOR bitfield
    DESCRIPTOR far *dt = MK_FP((seg&4)?0x68:0x8, 0);
  // in DOS/16M, selector 0x08 is GDT; selector 0x68 is LDT
    ACCESS_RIGHTS *pacc = (ACCESS_RIGHTS *) &dt[index].access;
▶   printf("SEL=%04X ADDR=%02X%04X LIMIT=%04X ACCESS=%d%c%c%c%c%c%c\n",
        seg, dt[index].addr_hi, dt[index].addr_lo,
        dt[index].limit,
        pacc->dpl,
        pacc->accessed ? 'a' : '-',
      access = 'ô' (0x93);
      reserved = 0;}
# dt[seg >> 3]
   struct  at 0008:0028 {
      limit = 255 (0xFF);
      addr_lo = 10240 (0x2800);
      addr_hi = '\002';
      access = 'ô' (0x93);
      reserved = 0;}
#
```

Aside from running in protected mode and consequently having the ability to handle much larger programs, its "immediate mode" is what most sets IC apart from other integrated development environments like Borland's Turbo C or Microsoft's Quick C. There is a fundamental difference between fully interactive C on the one hand and a merely fast C like Turbo C or Quick C on the other (though Turbo C and Quick C do produce *much* faster code than IC).

In many ways, IC represents the interactive style of languages like Forth and Lisp, made available for the C programming language. But, contrary to the stereotype of an interpreter, IC uses native object code. In fact, IC can dynamically load and link .OBJ and .LIB files, and can write out stand-alone .EXE files. Such stand-alone executables include a built-in protected-mode DOS extender.

The two major benefits of protected mode—memory protection and a large address space—mesh with the needs of a C development environment. The large address space (up to 16 megabytes of memory) means that even very large C programs can be developed interactively. Hardware-based memory protection helps insulate IC from bugs in user code, and assists IC in finding bugs. An interpreter running in protected mode can off-load some of its type-checking onto the CPU.

IC also illustrates an important trend in DOS extender technology: moving DOS extenders into language compilers. Other examples of this trend are the inclusion of Eclipse Computer Solutions' OS/286 in Lahey Fortran F77L-EM/16 and in Gold Hill Lisp, and its forthcoming inclusion in Lattice C. IC is an unusual and striking example of this trend.

IC is not only a product built using a DOS extender, it is also an example of why "Extended DOS" is necessary in the first place. IC has been in existence since 1984. As features were added to the product, it began to strain against the artificial 640K "Berlin Wall" of real-mode MS-DOS. Rational Systems in fact developed DOS/16M for IC, to cope with its expanding features and resulting expanding memory consumption. Thus, DOS/16M is based on IC, as much as IC is based on DOS/16M.

For a short time, Rational Systems marketed a separate protected-mode IC/16M alongside its real-mode IC. For an even shorter time, the name was changed to IC/PM but unfortunately this sounded like a reference either to OS/2's Presentation Manager or to the CP/M operating system! In October 1989, with IC version 4.0, Rational Systems discontinued the real-mode version.

There's one more reason to use IC as a tool for exploring protected mode: price. At $795, IC is not cheap, but it is more affordable than the $5,000 that Rational Systems charges for DOS/16M. It should be noted that IC and DOS/16M are totally different products. IC is not a scaled-down version of DOS/16M, nor is it intended to be used by itself as a DOS extender. For commercial software houses that need to move their products to protected-mode MS-DOS, the cost of DOS/16M might well be insignificant. But because it is both interactive and based on a DOS extender, IC does provide an excellent base to explore protected-mode programming.

Running LLIBCE.LIB in Protected Mode

Out of the box, IC runs in medium model; its data pointers are 2-byte quantities that hold only an offset. IC supports the Microsoft `near` and `far` keywords, but rather than explicitly specify `void far *` when we need more than 64K of data or require 4-byte data pointers, a large-model version of IC is useful.

Right away we encounter an unusual feature of IC: its ability to create new versions of itself. IC's `#savemod` command writes out a new version of IC.EXE, built-in DOS extender and all. Such a feature seems unusual in a C compiler, but IC is almost a full-blown protected-mode linker, and its ability to clone new versions of itself is a subset of this larger capability.

Working with a "quick" environment always raises the issue of compatibility with production compilers. IC has a standard library, but if you would rather, say, use the Microsoft C library, IC has scripts to: load in MSC 5.1's real-mode, large-model LLIBCE.LIB; load in some IC-supplied .LIB modules to replace the few Microsoft functions that don't work in protected mode; load the Microsoft `#include` files into IC; and then write out a new, large-model MSC-compatible IC. The new executable not only runs your C programs in protected mode under MS-DOS, it executes the Microsoft C library in protected mode too.

It seems like quite an accomplishment to load real-mode object code and execute it in protected mode, but, as we've seen, this is standard procedure for 16-bit DOS extenders such as DOS/16M and OS/286. Some MSC 5.1 calls are not currently supported by IC, including `_heapwalk()` and the graphics library.

Having built a large-model MSC 5.1-compatible version of IC with `#savemod`, it is also useful to add the DOS/16M library to IC.EXE, using the `#load` command. The names `#load` and `#savemod` show how IC provides a command language in the form of C preprocessor statements. The # can be left off most of these commands, but it indicates how these commands behave like preprocessor statements. In addition to the DOS/16M library, we also add the macros `FP_SEG`, `FP_OFF`, and `MK_FP`:

```
# #load dos16lib.c
// ...
# #defineg MK_FP(seg,ofs)   \
#    ((void far *) (((unsigned long) (seg) << 16) | (ofs)))
# #defineg FP_OFF(fp) ((unsigned short) (fp))
# #defineg FP_SEG(fp)   \
#    ((unsigned short) (((unsigned long) (fp)) >> 16))
# #savemod ic.exe
IC.EXE: 615K bytes in file.
```

```
# D16ExtAvail() + D16LowAvail()
    776240 (0xBD830)
# #quit
```

The #defineg command is like the C preprocessor #define statement, but defines a preprocessor symbol *globally*, across all .C files. This is another example of how a compiler-oriented language like C changes in an interactive environment.

Evaluating the expression D16ExtAvail() + D16LowAvail(), we see that on a 2-megabyte machine with several TSRs loaded, there is still 776K free, even with the inclusion of the Microsoft library, the DOS/16M library, and the IC environment itself. Because DOS/16M has an extremely small low-memory footprint and its default strategy is to allocate first out of extended memory, most of the lower 640K is free. On my computer there is 572K free low memory before entering IC; how much is free inside IC?

```
# D16LowAvail()
    525168 (0x80370)
```

It is practical to execute a text editor from within IC (though IC does come with a decent text editor built in), and even run another compiler or an assembler to produce an .OBJ file that IC can then load. This book was worked on from within IC:

```
# void e(char *s) {
>   char cmd[80];
>   sprintf(cmd, "epsilon %s", s);
>   system(cmd);                      // pass cmd to MS-DOS
>   if (strstr(s, ".c")) {            // after editing a C file...
>       sprintf(cmd, "#load %s", s);  // #load it
>       _interpret(cmd);              // pass cmd to IC
>       }
>   }
e defined.
# void book() { e("\\extdos\\chap04.doc"); }
book defined.
# book()
```

In this interactive environment, the C programming language also becomes a powerful "macro" command language. Since the programming language and command language are one and the same, there isn't the usual debugger problem of having to learn a new command language.

GP Faults and the Protected-Mode Interpreter

The interactive style of Instant-C is useful when you want to test whether a C expression is legal in protected mode and, in particular, in DOS/16M. For example, even if you don't have DOS/16M itself, you could use IC to try out the TICKS() macro given earlier. You don't need a program, and you don't need main():

```
# #define TICKS()   *((unsigned long far *) 0x46c)
# #define SECONDS() (TICKS() / 18)      // since midnight
# SECONDS()
 ## 534: Reference through null far pointer in command line
# #reset
# #define TICKS()   *((unsigned long far *) 0x0040006c)
# SECONDS()
   70957 (0x1152D)
```

Here, IC saw that the first version of TICKS() dereferenced the null pointer. The more interesting case is when protection violations are detected, not by IC, but by the CPU itself. For these, IC displays a different message. Take the example of the program BAD.C, which caused a GP fault when the user neglected to supply a command-line argument:

```
# load bad.c
STDIO.H included.
main defined.
BAD.C loaded.
# run 100
100
# run
 ## 492: Invalid address 0A80:4552 at __CATOX+000E
```

When the CPU detects a protection violation, it doesn't shut down the offending application. In fact, it wouldn't know how to: terminating applications is an operating system's business! All the processor does is issue an Int 0DH (GP fault). This is an interrupt like any other, and someone has to install a handler for it. Protected-mode operating systems like OS/2, and DOS extenders like DOS/16M, install Int 0DH handlers that respond to GP faults by shutting down the offending application. We saw an example of this earlier, when a version of the LIST program was shut down by DOS/16M for trying to use the bad version of the TICKS() macro.

This presents a problem for a protected-mode interpreter. The operating environment's GP fault handler just knows that an application caused a GP fault. The DOS/16M GP fault handler doesn't know that IC is an interpreter, and that its GP faults are actually caused by user input, not by the interpreter itself. If this default `INT 0DH` handler were in effect, IC itself would get shut down every time we typed in a protection violation!

But IC doesn't shut down when we commit a GP fault. Instead, *our* code is squelched, while the interpreter stays in control. This is exactly the behavior one wants in a protected-mode interpreter. That we can freely type protection violations in at the # prompt is due to the fact that IC installs its own `Int 0DH` handler. Instead of DOS/16M's default GP fault handler, IC's handler catches the signals that the CPU sends. (For a lengthy discussion of this issue, see "Stalking GP Faults," *Dr. Dobb's Journal*, January 1990 [Part 1] and February 1990 [Part 2].)

IC indicated that a GP fault occurred at `_CATOX+000E`. Unfortunately, IC currently doesn't have an object-code disassembler (it really needs to be more like the DOS/16M debugger), but it seems pretty obvious that `_CATOX()` was called from `atoi()`. If in doubt, we could use the IC `#backtrace`:

```
# back
Level 1 condition:  492: Invalid address 0A80:4552 at __CATOX+000E
  occurred in function ##unknown##
  called from function  atoi" line 0:
=>       393: source code unavailable
  called from function  main" line 3:
        {
=>          printf("%d\n", atoi(argv[1]));
        }
  called from function in object code
```

Anyway, the protection violation alerts us to the fact that our code needs fixing. We can use the editor built into IC, rerun the test, and save the source code back out to a file:

```
# #ed main
// use IC editor to add test: if (argc > 1)
# #run
// no GP fault now
# #save good.c
GOOD.C: 180 chars; 11 lines
```

The `#save` command will reformat your source code, according to formatting options set by assigning to variables such as `_braceundent`, `_KRbrace`, and so on.

Using Protection

In addition to helping find bugs during development, the hardware-based memory protection of the Intel processors can be put to work in the deliverable version of a product as well.

For example, functions often perform range checking. Each time the function is called, it checks its parameters against the size of the target object. The size may have been stored in the object header.

Since the hardware does range and type checking in protected mode anyway, and since we pay a performance penalty for this checking, we should get the hardware to do *our* checking as well. You've already paid for protection; you might as well get the most out of it.

This requires devoting a separate selector to each object for which you want hardware-assisted checking. To see why, let's overstep the bounds of an array and see whether the Intel processor detects an off-by-one fencepost error:

```
# char *p;
# p = malloc(2013)
    address 03B8:01A6: ""
# p[2013] = 'x'
    'x' (0x78)
```

The pointer p points to a block of 2,013 bytes, numbered 0 through 2012, so p[2013] clearly oversteps its bounds. If protected mode is all it's cracked up to be, the CPU should have complained. Why didn't it?

The reason is that `malloc()` and other high-level-language memory allocators suballocate out of pools of storage. They do not ask the operating system for memory each time you ask them for memory, nor would you want them to. Out of one segment, `malloc()` may allocate several different objects. While we think of the object p as containing 2,013 bytes, the processor sees a considerably larger object: the block of memory `malloc()` received the last time it asked DOS for memory. What is the object size the CPU sees?

```
# D16SegLimit(p)
    24575 (0x5FFF)
```

If this explanation is correct, trying to poke `p[0x5FFF]` had better cause a GP fault:

```
# p[0x5fff] = 'x'
 ## 492: Invalid address 0BF8:002A in command line
```

Now, we need a way to make the CPU see things *our* way. Since 80286 memory protection is based on segmentation, we must devote a separate selector to each object for which we want hardware-assisted checking. Notice I said *selector* and not *segment*. We can continue to use `malloc()` to allocate objects, but when we want the CPU to know how big we think the object is, we provide an *alias* in the form of another selector which points at the same physical memory, but whose limit is smaller. The segment limit indicates the highest legal offset within a block of memory and is checked by the CPU for each memory access.

To create an alias q for the pointer p, where q's limit is equivalent to the array bounds, we can use two other DOS/16M functions:

```
# char *q;
# q = D16SegAbsolute(D16AbsAddress(p), 2013);
   0C08:0000
```

This takes the physical address returned by `D16AbsAddress()`, together with the limit we're imposing for access to this memory, and passes them to `D16SegAbsolute()`, which constructs a protected-mode selector for the same absolute physical address but with a different limit. Did it work?

```
# D16AbsAddress(p) == D16AbsAddress(q)
   1
# D16SegLimit(p) == D16SegLimit(q)
   0
# D16SegLimit(p)
   24575 (0x5FFF)
# D16SegLimit(q)
   2012 (0x7DC)
# q[2013]
 ## 492: Invalid address 0BF8:002A in command line
```

Notice that attempting even to *read* from this invalid array index now causes a GP fault. We can dispense with explicit bounds checking; the CPU will check for us. To control the error message displayed when a GP fault occurs, we could write our own `INT 0DH` handler and install it using the DOS set-vector function (`INT 21H AH=25H`). Thus, instead of littering error checking throughout our code, protected mode allows us to centralize it in an interrupt handler. Errors can be handled after the fact, rather than up-front. In a way, this resembles ON ERROR, one of the more powerful concepts in BASIC (which got it from PL/I).

This meshes with the advice given by advocates of object-oriented programming: "If you are expecting a sermon telling you to improve your software's reliability by adding a lot of consistency checks, you are in for a few surprises. I suggest that one should usually check less. . . . defensive programming is a dangerous practice that defeats the very purpose it tries to achieve" (Bertrand Meyer, "Writing Correct Software," *Dr. Dobb's Journal*, December 1989).

By using protection, you may be able to make an application run faster in protected mode than under real mode, since a lot of error-checking and "paranoia" code can be made unnecessary in protected mode.

When finished with the pointer p, it is important not only to `free(p)`, but also to release the alias in q. Don't use `free()` to release this selector, though. The C `malloc()` manager doesn't know anything about q and, in any case, q is just an alias, a slot in a protected-mode descriptor table. We need to free up this slot because, as noted earlier, the number of selectors available in protected mode is quite limited:

```
    # free(p)
    # D16SegCancel(q)
```

In moving from real to protected mode, programmers may regret that segment arithmetic is so restricted. However, the ability to create aliases, different "views" of the same block of physical memory, means that protected-mode selector manipulation is actually far *more* versatile than real-mode segment arithmetic.

The Intel 286 Protected-Mode Instructions

While transparency is a major goal of "Extended DOS," sometimes it is useful to not be so transparent. For example, DOS extender diagnostic programs and DOS extender utilities will generally be non-portable, hyper-aware that they are running in protected mode.

The 80286, and also the 286-compatible 386 and 486 chips, have a number of instructions that Intel provides primarily for use by protected-mode operating systems, but which are also useful for utilities and diagnostic programs:

- `LSL` (load segment limit)—size of a segment
- `LAR` (load access rights)—access rights of segment
- `VERR` (verify read)—can segment be peeked?
- `VERW` (verify write)—can segment be poked?
- `SGDT` (store GDT)—base address and size of GDT
- `SIDT` (store IDT)—base address and size of IDT
- `SLDT` (store LDT)—selector to LDT.

For example, in the last section we called `D16SegLimit()` to find the size of the segments pointed to by p and q. In operation (though not in implementation), `D16SegLimit()` corresponds to the `LSL` instruction, which takes a selector in the source operand and, if the selector is valid, returns its limit (size - 1) in the destination operand. For example:

```
        lsl ax, [bp+6]
```

Similarly, the `LAR` instruction will load the destination operand with the *access rights* of the selector in the source operand, if it contains a valid selector:

```
        lar ax, [bp+6]
```

The instructions `LSL`, `LAR`, `VERR`, and `VERW` are special because, even if the selector in the source operand is *not* valid, the instructions don't GP fault; instead, the zero flag is cleared. This means that, if these instructions were available in a high-level language, we could construct protected-mode memory browsers and other utilities simply by looping over all *possible* selectors. This is an odd form of segment arithmetic:

```
        for (unsigned i=0; i<0xFFFF; i++)
            if (lar(i) is valid)
                print_selector(i)
```

It is easy to make the Intel protected-mode instructions available to C and other high-level languages, and they can be used interactively in IC. PROTMODE.ASM is a small library of functions, including `lsl()` and `lar()`, that can be assembled into PROTMODE.OBJ using either the Microsoft Assembler (version 5.0 and higher) or Turbo Assembler:

```
;       protmode.asm -- 286 protected-mode instructions
;       requires MASM 5.0 or higher or TASM
;       masm -ml protmode;
;       or, tasm -ml protmode;

        dosseg
        .286p
        .model large
        .code

        public  _lsl, _lar, _verr, _verw, _sgdt, _sidt, _sldt

;       extern unsigned far lsl(unsigned short sel);
;       input:   selector
;       output:  if valid and visible at current protection level,
;                   return segment limit (which is 0 for 1-byte seg!)
;                else
;                       return 0
;
_lsl    proc
        enter   0, 0
        sub     ax, ax
        lsl     ax, [bp+6]
        leave
        ret
_lsl    endp
;
;       extern unsigned short far lar(unsigned short sel);
;       input:    selector
;       output:  if valid and visible at current protection level,
;                       return access rights (which will never be 0)
;                    else
;                       return 0
;
_lar    proc
        enter   0, 0
        sub     ax, ax
        lar     ax, [bp+6]
        shr     ax, 8
        leave
        ret
_lar    endp
;
;       extern BOOL far verr(unsigned short sel);
;       input:    selector
;       output:  valid for reading ? 1 : 0
;
_verr   proc
        enter   0, 0
        mov     ax, 1
        verr    word ptr [bp+6]
        je      short verr_okay
        dec     ax
verr_okay:
        leave
        ret
_verr   endp
;
;       extern BOOL far verw(unsigned short sel);
;       input:    selector
;       output:  valid for writing ? 1 : 0
;
```

```
_verw   proc
        enter   0, 0
        mov     ax, 1
        verw    word ptr [bp+6]
        je      short verw_okay
        dec     ax
verw_okay:
        leave
        ret
_verw   endp
;
;       extern void far sgdt(void far *gdt);
;       input:   far ptr to 6-byte structure
;       output:  fills structure with GDTR
;
_sgdt   proc
        enter 0, 0
        les   bx, dword ptr [bp+6]
        sgdt  fword ptr es:[bx]
        leave
        ret
_sgdt   endp
;
;       extern void far sidt(void far *idt);
;       input:   far ptr to 6-byte structure
;       output:  fills structure with IDTR
;
_sidt   proc
        enter 0, 0
        les   bx, dword ptr [bp+6]
        sidt  fword ptr es:[bx]
        leave
        ret
_sidt   endp
;
;       extern unsigned short sldt(void);
;       input:   none
;       output:  Local Descriptor Table register (LDTR)
;
_sldt   proc
        sldt  ax
        ret
_sldt   endp

        end
```

Note that PROTMODE.ASM uses the `DOSSEG` directive, which simplifies writing assembly-language subroutines, and uses the `ENTER` and `LEAVE` instructions, provided on the 80286 and higher for working with high-level-language stack frames. These execute a little slower than the corresponding `PUSH BP / MOV BP, SP` prolog and `POP BP` epilog, but provide more compact source code.

If your C compiler has an in-line assembler facility, rather than use `PROTMODE.ASM`, you can instead place these functions inside a .C file. For example, using Microsoft C 6.0, `lsl()` can be defined as follows (compile with `-G2`):

```
unsigned lsl(unsigned sel)
{
    _asm sub ax, ax
    _asm lsl ax, sel
}
```

PROTMODE.ASM provides nothing more than a functional interface to the Intel protected-mode instructions. While completely non-portable with real mode, this module is highly portable among 16-bit protected-mode systems (it would require some modification for use with a 32-bit DOS extender). Once assembled into PROTMODE.OBJ, it can be linked into any 16-bit protected-mode program (including an OS/2 program). It can be loaded into IC with the following command:

```
        #loadobj "protmode.obj"
```

IC can also load .LIB modules, using the `#loadlib` command (this is how an MSC-compatible version of IC is produced). The object code can be produced by an assembler (as in the example of PROTMODE.OBJ), or by another C compiler. One unfortunate limitation is that IC attaches no meaning to the `pascal` keyword, so object code that uses the Pascal calling convention cannot be successfully called from within IC.

You need to supply stub definitions for the individual routines in an object module loaded into IC. These look almost like declarations or function prototypes, except that they are followed by the construct `{extern;}`. For example, after loading PROTMODE.OBJ, `lsl()` would be defined to IC as:

```
        unsigned lsl(unsigned short sel) {extern;}
```

PROTMODE.H is a C `#include` file that contains function prototypes for use with IC (`#ifdef InstantC`) or any other 16-bit protected-mode environment:

```
/* PROTMODE.H */
typedef enum { FALSE, TRUE } BOOL;
#ifdef InstantC
unsigned far lsl(unsigned short sel)           {extern;}
unsigned short far lar(unsigned short sel)     {extern;}
BOOL far verr(unsigned short sel)              {extern;}
BOOL far verw(unsigned short sel)              {extern;}
void far sgdt(void far *gdt)                   {extern;}
void far sidt(void far *idt)                   {extern;}
unsigned short sldt(void)                      {extern;}
#else
extern unsigned far lsl(unsigned short sel);
extern unsigned short far lar(unsigned short sel);
extern BOOL far verr(unsigned short sel);
extern BOOL far verw(unsigned short sel);
extern void far sgdt(void far *gdt);
extern void far sidt(void far *idt);
extern unsigned short sldt(void);
#endif
```

Now it's time to test the functions. Let's allocate a 10K segment, and see what limit `lsl()` returns:

```
        # char *p;
        # p = D16MemAlloc(10240)
            address 0C08:0000
        # lsl(FP_SEG(p))
            10239 (0x27FF)
```

Clearly, 10,239 is the last legal offset within a 10K segment, so `lsl()` seems to work. (Actually, there is an obscure bug in `lsl()`. Can you spot it? Hint: what is the `LSL` for a one-byte segment? For an invalid segment?)

The `verw()` function, like the `VERW` instruction, returns TRUE if a selector can be written to, or FALSE if the selector is read-only:

```
        # verw(FP_SEG(p))
            1
```

We can use a DOS/16M function to mark this segment as read-only, and then see if `verw()` has picked up on the change in the selector attributes:

```
        # D16SegProtect(p, 1)
            0
        # verw(FP_SEG(p))
            0
```

As the IC manual explains, `D16SegProtect()` can be used if you have some data that is being clobbered, or if you wish to ensure that only certain functions can update some data structure. It is important to note that the read-only attribute, like other aspects of the protected-mode access rights, applies to a *selector*, not to the underlying block of memory. One selector can be read-only and another read/write, while both correspond to the same physical memory.

Having tested the PROTMODE.OBJ routines we can, as promised, write a simple loop to display all valid selectors within our program. In IC, of course, we can just type this in at the # prompt:

```
        unsigned i;
        for (i=0; i<0xFFFF; i++)      // for all possible selectors
            if (lar(i))               // if a valid selector
                printf("%04X\n", i);  // print selector
```

This will display all valid selectors within a protected-mode program (not just a DOS/16M program). But to be genuinely useful we need to print out some additional information about the selectors:

```
/* BROWSE.C */

#ifdef InstantC
#loadobj "protmode.obj"
#endif
#include "protmode.h"

void browse()
{
    unsigned long addr;
    unsigned i, acc;
    for (i=0; i<0xFFFF; i++)              // for all possible selectors
        if (acc = lar(i))                 // if a valid selector
        {
            addr = D16AbsAddress(MK_FP(i,0));
            printf("%04X %06lX LAR=%02X LSL=%04X PL=%02X %s %s %s %s\n",
                i,                                // selector
                addr,                             // physical base addr
                acc,                              // access-rights byte
                lsl(i),                           // segment limit
                i & 3,                            // protection level
                verr(i) ? "VERR" : "    ",        // readable?
                verw(i) ? "VERW" : "    ",        // writable?
                i & 4 ? "LDT" : "GDT",            // which table?
                i == addr >> 4 ? "TRANS" : ""); // transparent?
        }
}
```

In addition to using several of the functions in PROTMODE.ASM, the code in BROWSE.C also performs some manipulations on the selector number itself: the bottom two bits are extracted with the expression `i & 3`, and the third bit is extracted with the expression `i & 4`.

It was noted earlier that a protected-mode selector is almost, but not exactly, a "magic cookie," in that the number itself actually has semantic meaning. Now all can be told: a selector is comprised of three fields. The bottom two bits contain a protection level, zero (most privileged) through three (least privileged). The third bit from the right contains a *table indicator*—zero means the selector belongs to the GDT, and one means it belongs to the LDT—and the remaining 13 bits form an index into this table.

Running under IC, a small part of the output from BROWSE.C looks like this:

```
0038 000000 LAR=93 LSL=FFFF PL=00 VERR VERW GDT
003C 000000 LAR=93 LSL=FFFF PL=00 VERR VERW LDT
0040 000400 LAR=93 LSL=0FFF PL=00 VERR VERW GDT TRANS
0044 000400 LAR=93 LSL=0FFF PL=00 VERR VERW LDT
0048 034FE0 LAR=93 LSL=FFFF PL=00 VERR VERW GDT
004C 034FE0 LAR=93 LSL=FFFF PL=00 VERR VERW LDT
0050 110010 LAR=93 LSL=200F PL=00 VERR VERW GDT
0054 110010 LAR=93 LSL=200F PL=00 VERR VERW LDT
0058 032050 LAR=81 LSL=0067 PL=00           GDT
005C 032050 LAR=81 LSL=0067 PL=00           LDT
0060 FA0000 LAR=93 LSL=FFFF PL=00 VERR VERW GDT
0064 FA0000 LAR=93 LSL=FFFF PL=00 VERR VERW LDT
0068 100010 LAR=82 LSL=FFF8 PL=00           GDT
```

The list runs on for quite a while.

What is the value of this? In contrast to real mode where every address you can form points *somewhere*, protected-mode memory is a sparse matrix. At any given time, most segment:offset combinations are not valid addresses—dereferencing them causes a protection violation. Producing a list like this gives us an idea of the memory organization of a DOS extender program.

From this list, we can see that while protected-mode memory is a sparse matrix, under DOS/16M it's not as sparse as under OS/2. From the vast number of valid selectors displayed by this program, it is obvious that IC is compiled with the full 64K GDT available under DOS/16M (the size of the GDT can be controlled with the MAKEPM utility).

We can also see that all the entries are marked PL=00, indicating that everything is running at Ring 0. To double check that this is so, the following code represents the query, *Are any segments not at Ring 0?*:

```
    for (i=0; i<0xFFFF; i++)
        if (lar(i) && (i & 3))
            printf("%04X PL=%02X\n", i, i & 3);
```

Under IC, this produces no output. Everything is running at the most privileged protection level. This is one of the differences between DOS/16M and a full-blown protected-mode operating system like OS/2. Because DOS/16M is just a shell to support one program at a time in protected mode, Rational Systems chose not to establish different protection levels.

Along the same lines, the selectors you'll use in IC or in DOS/16M actually refer, not to your program's LDT, but instead to the GDT. Since there is only one program running, the distinction between GDT and LDT, while crucial in a multitasking operating system like OS/2, is fairly artificial in the "one program at a time" world of DOS/16M.

On the other hand, Eclipse's OS/286, while sharing many of the same goals as DOS/16M, makes a sharper distinction between the kernel (the OS/286 DOS extender itself) and the program supported by the DOS extender. OS/286 programs run at Ring 3, while OS/286 itself runs at Ring 0. This just shows that there are few fixed rules about how a DOS extender *must* be organized; protected mode allows for a wide variety of styles in operating environments.

In a version of DOS/16M that complies with Microsoft's planned DOS Protected Mode Interface (DPMI), DOS/16M programs will more closely resemble OS/286 in that they will have to use the LDT, and will have to run at Ring 3. This, in turn, means that the only available *transparent* selector may be 40H. Rational Systems expects only about one-half of its clients to switch to this DPMI-compliant version.

IC requires a large GDT partially to support a large number of transparent selectors. In the example above, selector 40H has a physical base address of 400H, corresponding to the BIOS data area. Using the same code from PROTMODE.ASM, it is trivial to form the query, *Which selectors are transparent?*:

```
    for (i=0; i<0xFFFF; i++)
        if (lar(i) && (i == D16AbsAddress(MK_FP(i,0)) >> 4))
            printf("%04X ", i);
```

The preceding discussion of the three fields that make up a protected-mode selector, along with the discussion of transparent selectors, indicates that transparent selectors will generally come out of the GDT. Since the physical base address dictates the number of a transparent selector, since this number is merely a representation of the three fields, and since a zero in bit 3 indicates the GDT, it follows that whenever we want a transparent selector to a real-mode segment such as `40H` or `B800H`, it *must* belong to the GDT. This also applies to OS/2 bimodal pointers.

Examining the Protected-Mode Descriptor Tables

We have already used an indirect method to examine the DOS/16M memory map: loop over all possible selectors and see if they're legal. We can also directly examine the GDT, IDT, and LDT.

PROTMODE.ASM contains a functional interface to the `SGDT` instruction, which returns the physical base address of the GDT. `SGDT` expects a pointer to 6 bytes of storage (an FWORD PTR), into which it copies the contents of the CPU's GDT register (GDTR). The GDTR holds the 24-bit physical base address and 16-bit limit of the GDT, and corresponds to the following C structure (most of the structures that follow must be byte-aligned; in IC, set `_struct_alignment=1`, in batch compilers like Microsoft C, use `#pragma pack(1)`):

```
typedef struct {
    unsigned        limit, lo;
    unsigned char hi, reserved;
    } GDTR;
```

This structure can be used in IC, along with `sgdt()`:

```
# GDTR g;
# sgdt(&g)
# g
   struct  at 2F1C {
      limit = 65528 (0xFFF8);
      lo = 16 (0x10);
      hi = '\020' (0x10);
      reserved = '\0';}
```

Now that we have the physical base address (`100010H`) and limit (`FFF8H`) of the GDT, we need to map this into our address space. A protected-mode descriptor table is an array of 8-byte segment descriptors. Each descriptor contains the 24-bit physical base address and 16-bit limit for the segment, as well as an access-rights byte. There is also a 2-byte field used in 32-bit protected mode on the 386. All this can be expressed in C:

```
typedef struct {
    unsigned        limit;    // size minus 1
    unsigned        addr_lo;  // physical base addr - paragraph.byte
    unsigned char addr_hi;  // physical base addr - megabyte
```

```
        unsigned char access;    // see ACCESS_RIGHTS below
        unsigned       reserved; // for 386 (32-bit)
        } DESCRIPTOR;
```

After typing or loading this structure definition into IC, we can create a pointer to the GDT:

```
# DESCRIPTOR *gdt;        // GDT is array of DESCRIPTOR
# gdt = D16SegAbsolute((long) MK_FP(g.hi, g.lo), g.limit + 1)
    address 0C08:0000
```

Now we have a pointer to the GDT, let's make it read-only to make sure we don't mess anything up (though if you were working in a protected-mode environment that didn't have convenient functions for changing selector attributes, you might actually *want* to write to the GDT!):

```
# D16SegProtect(gdt, 1);
```

If bit 3 of a selector indicates that it belongs to the GDT, then the top 13 bits of the selector can be used as an index into the GDT. Take the example of the GDT pointer itself:

```
# gdt[FP_SEG(gdt) >> 3]
    struct  at 0C08:0C08 {
        limit = 65527 (0xFFF7);
        addr_lo = 16 (0x10);
        addr_hi = '\020' (0x10);
        access = 'Q' (0x91);
        reserved = 0;}
```

This confirms what we already know about the GDT: that its physical base address is 100010H and that its limit is FFF7H.

Now that we have a pointer to the GDT, we could dispense with such functions as D16SegAbsolute() and D16SegLimit(), and write portable protected-mode code. Data structures such as the GDT are a feature of protected mode itself, not of any environment in particular. Thus, while we're using DOS/16M and IC as examples here, almost all of this discussion really pertains to protected mode as a whole.

Since we can write portable protected-mode code once we have a pointer to the GDT, the question arises of whether there is a *portable* way to get this pointer. Unfortunately, there isn't. Getting its physical base address *is* portable, since this requires the Intel SGDT instruction, but this physical base address must then be mapped into an application's address space, and that will require some special facility within your protected-mode environment. Here, we used D16Seg-

`Absolute()`, which obviously won't work outside DOS/16M. But an equivalent function can always be found. For example, in OS/2 you would use the `PhysToUVirt` device driver helper function (`DevHlp`).

Returning to the GDT, let's examine the access-rights value which the CPU in protected mode uses to ensure proper use of a selector such as `0C08H`. We can use a C bitfield to display the individual fields that make up an access-rights value like `91H`. As noted when presenting a similar structure in Chapter 1, though, the ordering of bit fields is compiler-dependent; in addition, the following access-rights structure must be *byte*-aligned:

```
typedef struct access {
    unsigned accessed   : 1;  // has segment been accessed?
    unsigned read_write : 1;  // if data 1=write; if code 1=read
    unsigned conf_exp   : 1;  // expansion direction
    unsigned code_data  : 1;  // 0 = data, 1 = code
    unsigned xsystem    : 1;  // 0 = system descriptor
    unsigned dpl        : 2;  // protection level: 0..3
    unsigned present    : 1;  // is segment in memory?
    } ACCESS_RIGHTS;

# *((ACCESS_RIGHTS *) &gdt[FP_SEG(gdt) >> 3].access)
   struct access at 0C08:0C0D {
      accessed : 1 = 1;          // it's been used
      read_write : 1 = 0;        // it's read-only
      conf_exp : 1 = 0;          // it's not a stack
      code_data : 1 = 0;         // it's data
      xsystem : 1 = 1;           // it's not a system descriptor
      dpl : 2 = 0;               // protection level 0
      present : 1 = 1;}          // it's present in memory
```

All these data structures are described in the Intel literature on 286 and 386 protected mode. Seeing them come to life in IC, though, is a great aid to understanding protected mode.

One of the tricks of protected-mode programming is to acquire an in-depth knowledge of these data structures and then, when programming, to forget about them. The operating environment takes care of maintaining the GDT, and the descriptors within the GDT, and the access-rights bytes within the descriptors. The CPU will take care of using these data structures to maintain the integrity of the system. You're better off not thinking too closely about them, but it does help to have been familiar with them at some point or other.

While DOS/16M (and consequently IC) doesn't make much use of the LDT, this table is crucial in other protected-mode environments. To get a selector to your LDT, simply call `sldt()`. Though this is the LDT for your process, it may not be valid within your address space. It is simple to test if it is:

```
DESCRIPTOR far *ldt;
if (verr(sldt())
    ldt = MK_FP(sldt(), 0);
```

If the LDT selector is not valid within your address space, then you can instead look up its descriptor within the GDT:

```
DESCRIPTOR ldt_desc = gdt[sldt() >> 3];
```

and then get a pointer to the absolute address you find there, using whatever function is equivalent to `D16SegAbsolute`:

```
ldt = D16SegAbsolute((long) MK_FP(ldt_desc.hi, ldt_desc.lo),
    ldt_desc.limit + 1);
```

Now that we have these structures and have tested them, we can write a function to display selector attributes:

```
void sel(void far *fp)
{
    extern DESCRIPTOR far *gdt;
    extern DESCRIPTOR far *ldt;
    unsigned seg = FP_SEG(fp);
    unsigned index = seg >> 3;
    DESCRIPTOR far *dt = (seg & 4) ? gdt : ldt; // table indicator
    ACCESS_RIGHTS *pacc = (ACCESS_RIGHTS *) &dt[index].access;
    printf("SEL=%04X ADDR=%02X%04X LIMIT=%04X ACCESS=%d%c%c%c%c%c%c\n",
        seg, dt[index].addr_hi, dt[index].addr_lo, dt[index].limit,
        // display access rights as if they were file attributes:
        pacc->dpl,
        pacc->accessed ? 'a' : '-',
        pacc->read_write ? ((pacc->code_data) ? 'r' : 'w') : '-',
        pacc->conf_exp ? ((pacc->code_data) ? 'f' : 'e') : '-',
        pacc->code_data ? 'c' : 'd',
        pacc->xsystem ? '-' : 's',
        pacc->present ? 'p' : '-');
}
```

This code contains no references to DOS/16M. Once you have a pointer to the GDT and LDT, this function will work in any 16-bit protected-mode system.

We can test this function using some selectors hard-wired into DOS/16M:

- 00H—Null descriptor
- 08H—Global Descriptor Table (GDT)
- 10H—Interrupt Descriptor Table (IDT)

- 28H—Program Segment Prefix (PSP)
- 30H—DOS environment
- 38H—Descriptor for physical address 0
- 40H—Transparent descriptor for BIOS data area (paragraph 40)
- 68H—Local Descriptor Table (LDT).

For instance, selector 40H should point to physical address 400H, should be 4K (its limit should be 4,095 or FFFH), and it should be writable data:

```
# sel(MK_FP(0x40,0))
SEL=0040 ADDR=000400 LIMIT=0FFF ACCESS=0aw-d-p
```

In addition, we see that the selector has been accessed, is present in memory (IC currently does not include VM), and can be accessed only from Ring 0.

How about the GDT itself?

```
# sel(MK_FP(0x08,0))
SEL=0008 ADDR=100010 LIMIT=FFF8 ACCESS=0aw-d-p
```

The GDT is located at physical address 100010H; its limit is FFF8H. This is the maximum size for a GDT in 16-bit protected mode—there are 8,192 selectors.

Sel() can be used to examine the selector for any pointer, such as sel()'s own function pointer. The attributes display indicates this is readable code running at protection level zero:

```
# sel(sel)
SEL=0A68 ADDR=472BB0 LIMIT=43FF ACCESS=0ar-c-p
```

Finally, we can save all our structure and function definitions out to a file, using the IC #save command. The results can be read back in using the #load command.

IC as Protected-Mode Linker

A final feature of IC is the #make command, which creates stand-alone protected-mode executables. The resulting executables are rather large (the minimum size is 169K), and IC-compiled code is extremely slow (for compiled code; for interpreted code, it is very fast!), but since IC can load object code compiled by Microsoft C, including object code containing the function main(), IC could be used as a protected-mode linker. You have to produce an {extern;} declaration for each function. Here is an example, in which a tiny program is typed in IC, written out to a file with #save, compiled with Microsoft C, the compiled code loaded back into IC with #loadobj, and a protected-mode executable written out with #make:

```
C:\IC>ic
# void main(void) {
>    extern long D16LowAvail(), D16ExtAvail();
>    printf("Low memory: %lu\n", D16LowAvail());
>    printf("Extended memory: %lu\n", D16ExtAvail());
>    }
main defined.
# #save mem.c
MEM.C: 121 chars; 5 lines
# system("cl -AL -Ox -c mem.c");
Microsoft (R) C Optimizing Compiler Version 5.10
Copyright (c) Microsoft Corp 1984, 1985, 1986, 1987, 1988. All rights reserved

mem.c
# #delete main
# #loadobj mem
# void main(void) {extern;}
main defined.
# #make mem.exe
MEM.EXE: 186K bytes in file.
# #qquit
C:\IC>mem
DOS/16M Protected Mode Run-Time     Version 3.62
Copyright (C) 1987,1988,1989 by Rational Systems, Inc.
Low memory: 420384
Extended memory: 1023088
```

This is slightly cumbersome, but if IC were improved any more along these lines, it would start to compete with Rational Systems' other product, DOS/16M!

Chapter 5

80386-based Protected-Mode DOS Extenders

M. Steven Baker and Andrew Schulman

In the PC arena, new hardware outstrips software. It is common to find 80386 computers used merely as fast XT machines, because the truly powerful features of this CPU are unavailable in the real mode used by the venerable MS-DOS operating system. But as long as speed and power spur marketing, the developer can't neglect the powerful features of the 80386 and 80486 CPUs, even in the DOS marketplace.

386-based protected-mode DOS extenders grew out of the need to take advantage of the more powerful features of 80386 computers, without forgoing MS-DOS. 386 DOS extenders allow large applications to run under DOS until (if ever) a true 386 operating system displaces DOS.

These 386 DOS extenders set up a bridge to the DOS environment and put the 80386 chip into protected mode so that both the larger memory space and the full 386 instruction set are available. To an end user, an application built with a DOS extender can look like any other DOS program. When the user executes the program, control passes invisibly to the 386 DOS extender, which loads and runs the application in protected mode. Such a protected-mode application can transparently invoke real-mode DOS or BIOS services: 386 DOS extenders use essentially the same mechanism as was explained in detail in Chapter 4 (though a 386 DOS extender has the option of running DOS and BIOS in Virtual 8086 mode rather than in real mode). The application built with a DOS extender is likely to be much faster and have more features than a comparable DOS program, and it is often difficult or impossible to build a comparable real-mode DOS program.

In early 1987, when 386 DOS extenders were first introduced, many industry watchers predicted their demise within two years. Even the software developers who were marketing DOS extenders thought they had only a narrow "window of opportunity," as DOS would presumably soon be replaced by a true protected-mode operating system. By fall 1987, it seemed that MS-DOS, and DOS extenders, would have a much longer life, perhaps up to five years. Now, a few years later, 386 DOS extenders are solidly entrenched in the PC developer's market. MS-DOS, suitably extended, remains the operating system standard for the IBM microcomputer world, and the number of high-end commercial applications built with 386 DOS extenders continues to grow.

The best-known 32-bit DOS extender is Phar Lap Software's 386 | DOS-Extender. Applications built using the Phar Lap DOS extender include Interleaf Publisher (IBM), Mathematica (Wolfram Research), and AutoCAD 386 (Autodesk). In addition, Phar Lap defined the EASY OMF-386 format for 32-bit object files, and produces the industry-standard tools 386 | ASM and 386 | LINK, used even with their competitors' DOS extenders.

While this chapter deals primarily with Phar Lap's 386 | DOS-Extender, we will also examine OS/386 from Eclipse Computer Solutions. We have already said a good deal about the Eclipse product in Chapter 4 since, as noted there, OS/286 and OS/386 are tightly coupled. We will also briefly examine X-AM from Intelligent Graphics Corporation (IGC). While IGC is not currently marketing X-AM, it is incorporated in a number of important applications, including the dBase-compatible database manager FoxBase+/386 (Fox Software).

386 DOS Extenders in the Marketplace

The high-end CAD market on the PC is dominated by versions of applications incorporating 386 DOS extenders. These math-intensive, memory-intensive products benefit significantly from the 386 programming features made available by DOS extenders. The vendors in this market are compelled to offer 386 versions in order to remain competitive.

386 DOS extenders may be found in other graphics- and numerics-intensive applications ranging from symbolic math packages such as Mathematica to high-

end page printing programs such as Interleaf Publisher. Both of these products were ported from the UNIX workstation platform to DOS.

386 DOS extenders also appear in database products—most notably Paradox/386 and FoxBase+/386. A number of program development environments employ 386 DOS extenders, including Smalltalk-80/386 (ParcPlace), Common Lisp CLOE-386 (Symbolics), Laboratory Microsystems UR/Forth and APL-PLUS II (STSC).

Certainly, other schemes such as EMS can provide partial solutions to DOS memory limits. But 386 DOS extenders solve both memory and speed problems simply and directly. When your application outgrows memory space or needs a performance boost, it's time to consider 32-bit programming.

32 Bits

How is a 386-based protected-mode DOS extender different from the 286-based extenders we examined in Chapter 4? Remember that these 286 extenders, while based on AT class machines, can also run on the 386 and 486. Since a purely 386-based extender has the obvious disadvantage of addressing a smaller share of the market, what advantages does it have over a 286-based DOS extender?

All the advantages of using the 386's native mode can be summed up in the single phrase *32 bits*. If you have heard this phrase bandied about in the trade press, but have never seen a sample of 32-bit code, then you are in for a treat. Once you've used 32-bit code, you will never want to go back to the 16-bit code you've been using.

Let us take a somewhat contrived C function and examine the way it might be implemented, first using 16-bit code, and then using 32-bit code:

```
int foo(char *p)
{
    char *q;
    q = p;
    return *q;
}
```

Were you to compile this in small model (16-bit pointers) with a typical 16-bit MS-DOS compiler such as Borland's Turbo C, the resulting assembly output might look something like this:

```
mov si, word ptr _p          ; move p into an index register
mov al, byte ptr [si]        ; dereference the index register
cbw                          ; sign-extend AL into AX
```

This assembly language implementation closely matches the higher-level C representation. It is hard to improve on this code.

Unfortunately, most commercial software (word processors, spreadsheets, database managers, telecommunications programs, etc.) require more data space than the 64K maximum allowed by small model. Thus, commercial PC software is frequently compiled with the compact or large model, using 32-bit (four-byte) pointers in a 16-bit environment. Using, for example, another typical 16-bit DOS compiler (Microsoft C 5.1), the large-model implementation of `foo()` looks like this:

```
mov ax, word ptr _p            ; move bottom half of p into AX
mov dx, word ptr _p+2          ; move top half of p into DX
mov word ptr _q, ax            ; move AX into bottom half of q
mov word ptr _q+2, dx          ; move DX into top half of q
les bx, dword ptr [_q]         ; load far pointer into ES:BX
mov al, byte ptr es:[bx]       ; dereference ES:BX
cbw                            ; sign-extend AL into AX
```

What happened? Why did three simple C constructs swell into seven assembly-language statements? An inherent inefficiency of 16-bit code is revealed: 32-bit quantities such as longs (dwords) and far pointers are moved piecemeal, 16 bits (two bytes) at a time. Remember that these 32-bit quantities are the rule rather than the exception in commercial software. Also remember that running this code on the fastest 386 or 486 CPU will not make it transfer more than two bytes at a time. To do that, you need 32-bit protected mode: not just protected mode, mind you, but *32-bit* protected mode, since a 286-based DOS extender, even running on a 386 machine, is still very much a 16-bit beast.

Now for a breath of fresh air. Here is how `foo()` is implemented in flat model (four-byte pointers) by one 32-bit C compiler, Watcom C 7.0/386, that produces code suitable for a 386 DOS extender:

```
mov eax, _p                    ; move p into extended AX (EAX) register
movzx eax, byte ptr [eax]      ; dereference EAX, zero-extend into EAX
```

These two lines of 32-bit code illustrate many of the advantages of using a 386 machine as it was meant to be used, in protected mode, not as a fast 8088.

First of all, we see that, once one decides to use the full 32-bit registers on the 386 (such as EAX instead of AX), 32-bit quantities can obviously be MOVed into them in one fell swoop.

Second, having 32-bit registers opens the possibility of keeping 32-bit quantities in registers rather than on the stack.

Third, since the 386 allows dereferencing of almost any register, instead of only the old base (BX and BP) and index registers (SI and DI), a construct such as [EAX] can be used, instead of having to do something like MOV BX, AX followed by [BX]. This more flexible use of registers helps with the notorious "too few registers" problem faced by compiler writers.

Fourth, note how the LES BX instruction disappeared when we switched to 32-bit protected mode. In fact, all mention of segmentation disappeared entirely. Again, this provides a sharp contrast with a 286-based protected-mode version, which not only requires the LES BX instruction, but which additionally exacts a stiff penalty for its use, as noted in Table 4-2 in the previous chapter. Throughout this chapter, we will see that 32-bit protected-mode allows you to largely forget about segmentation.

Fifth, the 386 supports many new instructions, such as MOVZX in the example on the preceding page.

With all the advantages exhibited in this tiny example, it is not surprising that 32-bit code can easily execute much faster than comparable 16-bit code on the exact same hardware. The massive waste involved in using 386s as "fast" XTs should now be clear.

It should also be clear that, to reap these benefits, we cannot simply cannibalize the output of a 16-bit compiler as we did when using 286-based DOS extenders. Unless you are writing entirely in assembly language, using a 386 DOS extender requires that you switch to a 32-bit compiler, such as MetaWare High C-386, Watcom C/386, NDP Pascal-386, or Lahey FORTRAN F77/L32.

Now, it is true that the full 32-bit registers can be used in real mode as well. Few standard MS-DOS compilers provide an option to generate 80386 instructions (for example, Microsoft C has a -G2 switch to generate 286 instructions, but no equivalent -G3 switch), but, in parts of your program that will only run on a 386 or 486, you could include statements such as the following, which reads the four-byte BIOS timer count into EAX:

```
xor ax,ax                   ; zero ax
mov es,ax                   ; mov 0 into ES
mov eax, es:[46Ch]          ; dereference dword ptr 0000:046C into EAX
```

There are at least two limitations to this approach, however.

Multitasking software such as Windows or the OS/2 compatibility box, which use only the bottom 16 bits of the registers to save a program's context, can wreak havoc with real-mode programs that use 32-bit registers.

Second, when using 386 instructions in real mode, we are still stuck with the 640K limit of MS-DOS, and the one-megabyte limit of real mode itself. The benefit of 32-bit processing in 386 protected mode is not simply greater speed, but far greater space as well.

32-bit protected mode removes not only the 640K DOS barrier, but also the equally important 64K limit on segment size. Since 32-bit registers can be used as base and index registers, the near pointers loaded into these registers can use up to 32 bits for addressing memory. This in turn means that the maximum index within a memory segment is no longer `FFFFH` (64K), but `FFFFFFFFH` (four gigabytes).

Benefits of Using 386 Protected-Mode DOS Extenders

We have seen that, once a DOS extender opens up the power of the 80386/80486 CPU, a number of key features are available to the programmer. Let's now discuss the following benefits more systematically:

- large memory spaces for code and data
- powerful 32-bit instructions
- virtual memory options
- highly-optimizing compilers
- faster numerics using Weitek math coprocessors.

Wide Open Spaces

While the 8088 microprocessor used in the original IBM PC could address only 1 megabyte of physical memory, the 80386 and 80486 CPUs can access much larger memory spaces—up to 4 gigabytes. Typical 80386 AT and PS/2 style machines support up to 16 megabytes of physical memory, but some of the new EISA 80486 computers (for example, Compaq SystemPro) can support up to 256 megabytes of physical memory.

In this context, the 640K DOS memory limit is a barrier to developing large applications for the PC, or moving large applications to the PC. With the 640K DOS limit, large PC programs that depend on overlays and swapping code or data to disk are prevented from taking advantage of the larger memory capacity of the 386/486 PCs. Furthermore, the segmented architecture of 80X86 real mode still limits code and data segments to 64K, so 80386 instructions that would access memory without the inconvenience of 64K segment limits can't be used. The instructions include all the memory indirect and indexing instructions that use registers such as EBX with values greater than 64K, the maximum value of BX.

Because 32-bit protected mode breaks this 64K segment barrier in addition to the 640K DOS barrier, the most common memory model is a flat one (analogous to real-mode "tiny" model used in .COM files), in which all segment selectors point to the same block of memory—up to four gigabytes of 32-bit address space, with no segments. In a high-level language such as C, a 32-bit near pointer is a four-byte quantity. This in turn means that you almost never have to deal with segmented far pointers: once loaded, DS and CS can stay constant.

While it is no longer needed for an application's data and code, however, segmentation may be used to implement sharing and to enforce protection. In a DOS extender, segments are sometimes needed to use real-mode services. When you do need to specify a segment as well as an offset, the resulting far pointer is a six-byte quantity (an FWORD).

On those rare occasions when you have to change a segment register, the same penalty applies to 386 instruction times as we found in 286 protected mode (see Table 4-2).

32-bit Instructions

The 80386/486 microprocessors are full 32-bit CPUs that feature instruction sets much closer in power to the CPUs used in minicomputers than to older microprocessors like the 8086:

- Register and memory access is widened to 32 bits. 16-bit registers such as BX, BP, IP, and FLAGS, have been extended to 32-bit registers such as EBX, EBP, EIP, and EFLAGS.
- All 32-bit registers, except ESP, can be used as either base or index registers for memory addressing.
- A scaling factor (2, 4, or 8) can be applied to an index register for memory addressing.
- Two additional segment registers (FS and GS) have been added for addressing memory. Additional control registers (CR0, CR2, CR3), test registers (TR6, TR7), and debug registers (DR0, DR1, DR2, DR3, DR6, DR7) have been added.
- String instructions can now operate on double words (4 bytes).
- Instructions are available for converting an 8-bit or 16-bit operand to 32 or 64 bits (CWDE, CDQ, MOVSX, MOVZX).

- Bit manipulation instructions are added for testing, setting, and scanning bits (BT, BTC, BTR, BTS, BSF, BSR).
- The signed multiply (IMUL) instruction has a more general form that allows the use of any register for a destination.
- The LEA instruction is enhanced to perform fast integer multiplication.
- Instructions are added to set or clear bytes based on condition codes in the flags register (SETcc).
- 386 shift instructions support 32-bit and 64-bit shifts (SHLD, SHRD).

These instructions are available in 386 real mode as well as in protected mode. But in order to use the features of the 80386 within real-mode DOS, the developer would have to provide two versions of an application—one for the 8086 and one for the 80386. The preferred scheme would be to have the program sense the presence of the 80386 at runtime and use the faster 386 instructions available. To be most beneficial, the 80386 instructions would need to be programmed as in-line code. Only a few 80386 instructions can justify such effort in real mode. These instructions include MOVSD (double word move) and DIV and IDIV (long integer divide).

The true power of the 386 instruction set can be realized only when a program is targeted directly for 32-bit protected mode. Instructions now operate on 32-bit registers as well as the 8-bit and 16-bit registers of earlier Intel chips. The IDIV and DIV instructions were the slowest instructions on the 8086. In supporting the four-byte data type such as "long" in C, and INTEGER in Fortran, subroutines for addition, subtraction, division, and multiplication needed to be called. On the 80386, these subroutines can be replaced with single in-line instructions. In terms of clock cycles, long division shows the greatest benefit in execution speed. A long divide library routine is one place where using these 386 instructions in real mode justifies the effort to detect the 80386.

In the earlier Intel CPUs, memory could be addressed using the BP and BX registers as base pointers, and SI and DI registers for indexing. The BX register defaulted to addressing the data segment (DS), and the BP register defaulted to the stack segment (SS) for local (dynamic) storage. The 80386 makes memory addressing more general: any 32-bit register can be used as a base register. And all eight 32-bit registers, except ESP, can be used for indexing. In addition, a "scaling" factor of 2, 4, or 8 can be applied to any register used for indexing when referencing memory. This is a very attractive feature for indexing arrays of words (2-byte integers), double words (4-byte long integers and reals), quad words (8-byte double precision reals), and some multidimensional arrays. These addressing modes are well suited to the needs of high-level languages. In the following examples, note that the notation `12[edx]` is equivalent to `[edx+12]`:

```
mov esi, [eax]            ; using EAX as a base register

mov eax, 12[edx]          ; using EDX as base with displacement

mov eax, [ecx + edx*4]    ; using ECX as base pointer and
                          ; EDX as an index register with scaling
mov eax, 256[esp + edx*8] ; using ESP as base, EDX as index with
                          ; a displacement (stack segment)
```

The string instructions now support forms that operate on up to 32 bits at a time, and can use the EAX register rather than AX. These instructions include LODSD, STOSD, MOVSD, CMPSD, and SCASD. The MOVSD (move double from [ESI] to [EDI]) instruction executes in the same time as the earlier MOVSW (move word from [ESI] to [EDI]). When used with the REP (repeat) prefix operator for block moves, MOVSD is another instruction that is valuable to use in real mode and justifies the effort to detect the 386 chip. Since the ECX register is used for the loop counter, large blocks (up to 4 gigabytes) can be moved at one time:

```
cld                  ; set forward direction for block move
mov ecx, 28000h      ; count of double words to move
mov esi, source      ; point to source
mov edi, destination ;   and destination
rep movsd            ;   and move them
```

Several instructions are available for converting one operand size to a larger one. CWDE (convert word to dword, extended) sign-extends AX into EAX. CDQ (convert dword to quadword) sign-extends EAX into EDX:EAX. MOVSX is a more general form that sign-extends a byte or a word into a 16-bit or 32-bit register. MOVZX is a similar instruction that zero-extends into a wide register from either register or memory.

Six instructions are added to operate on bits in either registers or memory. BT (bit test) can be used to determine the setting of any arbitrary bit. For simple bit testing, the AND (logical) instruction can execute faster. But the BTC (bit test and complement), BTR (bit test and reset), and BTS (bit test and set) instructions combine bit testing with bit setting, clearing, and complementing, and are useful for implementing semaphores. BSF (bit scan forward) and BSR (bit scan reverse) find the first set bit (value of 1) in a bit stream. These instructions can be very useful when working on bit arrays, including graphics routines and allocation schemes for memory or disk space. A key feature is that these instructions can operate on bitmaps as large as 4 gigabits in length.

Several enhancements handle integer multiplication. IMUL (signed multiplication) is no longer limited to only the EAX register; any register can be the destination, providing greater flexibility. The fastest execution improvement is a new form of the LEA (Load Effective Address) instruction using scaled index addressing that performs fast integer multiplication in registers. This LEA form is limited to multiplications by small integers (2, 3, 4, 5, 8, and 9). However, this instruction executes in two clock cycles—far faster than either multiply or shift instructions—so a couple of LEA instructions can still execute faster than one regular multiply. The following example converts hours in the CH register to minutes in the EAX register:

```
; the traditional method
mov eax,60
mul ch

; using fast small integer multiplies
movzx eax,ch         ; start with hours
lea eax,[eax+eax*4]  ; x 5 use some fast integer multiplies
lea eax,[eax*4]      ; x 20
lea eax,[eax+eax*2]  ; x 60
```

A large group of SETcc (set byte on condition code) instructions are added to set or clear bytes based on boolean conditions. All conditions supported by the JMP instructions are allowed. These instructions provide a way to set logical variables, without using conditional JMPs that empty the prefetch instruction queue:

```
; the traditional method
xor edx,edx          ; assume boolean variable is false
or  eax,eax          ; test EAX for non-zero
jz  next             ; if zero, this JMP flushes the prefetch queue
inc edx              ; set EDX to true
next:

; using SETcc instruction
xor   edx,edx        ; assume this boolean variable is false
or    eax,eax        ;    and test for non-zero in EAX
setnz dl             ; set flag DL=true if EAX is non-zero
```

Paging and Virtual Memory

Along with the segmented memory management also found in 286-based protected mode, the 386 provides the ability to use paging, in which physical memory is tiled with 4K pages to form a linear address space. The use of paging and segmentation combines the best of both memory-management techniques; such a combination is not unique to the 386, and is described in many standard textbooks on computer architecture.

Using the hardware paging capabilities of the 80386/486, Phar Lap's 386 | DOS-Extender has a virtual-memory option, 386 | VMM (Virtual Memory Manager). As under more powerful operating systems like UNIX and OS/2, the amount of memory available for an application is limited by available disk space rather than by physical memory in the machine. The virtual memory feature allows you to run memory-hungry applications on 386 machines configured with relatively small amounts of physical memory. In one high-end technical publishing program ported from a workstation environment (Interleaf Publisher), the original version for the 386 required a minimum of 5 megabytes of physical memory to run. The same program incorporating the Phar Lap Virtual Memory Manager needs only a 2-megabyte machine.

Of course, you incur a performance penalty when you use virtual memory. If the entire application will not fit in physical memory, inevitably, some code or data is paged to a disk file. Depending on the organization of the program, code and data are automatically brought into memory as needed from disk. The less swapping that takes place, the faster the program executes. A program targeted for operation under a virtual memory manager will benefit from up-front planning and design to minimize swapping.

In addition, paging itself can also slow down memory access. Although the 80386 has a built-in, on-chip cache (the translation lookaside buffer—TLB) of the 32 most-recently-used page table entries to speed memory access, that represents only 128K of memory. Intel's simulations suggest that this should accommodate about 98 percent of normal memory access, but this depends on the code. If a page table entry is not in the cache, the 386 must read two double-word entries from the page translation table in memory before it can access the actual memory location of interest.

Optimizing Compilers

In addition to the language compilers moved up to the 386 from the DOS world, a number of highly optimizing compilers have been ported to the 386 from minicomputers and the UNIX workstations. Because the 80386 is a full 32-bit microprocessor, minicomputer vendors have been able to retarget their compilers to this CPU. These include the Numeric Data Processor (NDP) series from MicroWay (retargeting Green Hills compilers), and compilers from Silicon Valley Software (SVS) and Language Processors, Inc. (LPI).

Many of these compilers have a mainframe rather than a PC "feel." For instance, NDP FORTRAN-386 and SVS FORTRAN-386 are more likely to be used for porting an application "down" from the VAX to the 386, than for porting an application "up" from the PC.

Because these compilers run on the 386, both local and global optimizations are possible that would be difficult or impossible within DOS memory limits. The execution speed of the code generated from these optimizing compilers is most noticeable for math-intensive CAD and scientific applications.

Many compilers perform local optimizations within a function. However, global optimization across an entire source file requires keeping in memory (usually a tree structure) information concerning the entire source file's code. This tree structure can require large amounts of memory as the source file size grows. Under DOS' 640K limits, even local optimizations can be limited—unless the compiler pages its own tree structures to disk (MetaWare pages to disk in its DOS compilers). Global optimization on large Fortran source code files, for example, can become very difficult under DOS memory limits. With virtual memory on workstations, and now on 386 DOS extenders, compiler vendors are able to perform more complex local and global optimizations.

Not all 386 compilers are targeted for use with a DOS extender, though. A number of 386 compilers are designed to generate code for UNIX or other operating systems. For example, Intel itself markets compilers such as C-386/486 and FORTRAN-386, but these generate code for embedded applications, not for DOS extenders.

A 386 DOS Extender Application

Many of the benefits of 386 DOS extenders, as well as some of the mechanics of compiling, linking, and running a 32-bit protected-mode DOS application, are illustrated in the following C program, which can produce a very large bitmap of prime numbers, using the Sieve of Eratosthenes. The Sieve has a bad name due to its overuse in computer benchmarking, but this sieve is a little more interesting than most: it uses a bitmap rather than an array of integers; the bitmap is dense, in that multiples of the first two primes, 2 and 3, are neither computed nor stored.

The end result is a program that, running under the Phar Lap 386 | DOS-Extender on a 16 MHz Compaq Deskpro 386, takes only seven seconds to find the 78,498 primes <= 1,000,000, using a 41K bitmap. Since this sieve algorithm runs in linear time, and since the bitmap size also progresses linearly, you can extrapolate the time and space required to find p(*n*), the number of primes <= *n*, for any *n*. For instance, to find p(100,000,000) would take about 700 seconds and require a 4-megabyte bitmap (however, 80386/486 machines with static RAM caches will execute the sieve non-linearly for smaller values of *n*).

```
/*
SIEVE.C
Author: Andrew Schulman, February 1990
*/

#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <limits.h>
#include <time.h>

#include "bitmap.h"

#define N(x)                ((x) / 3)  // exclude multiples of 2 and 3

void fail(char *s)          { puts(s); exit(1); }

main(int argc, char *argv[])
{
    BITMAP map;
    FILE *f;
    double dsize;
    clock_t t1, t2;
    float runtime;
    ULONG i, j, n, primes, size, sqrt_size, map_size;
    int incr, jincr;

    if (argc  2)
        fail("syntax: [run386] sieve <x>");
    if ((dsize = strtod(argv[1], 0))  5)
        return 1;

    // estimate number of primes, using Legendre formula (1778)
    printf("Prime Number Theorem estimates: %.0f primes <= %.0f\n",
        floor(dsize / (log(dsize) - 1.08366)), dsize);

    if (dsize  ULONG_MAX)
        fail("number too large");
    size = (ULONG) dsize;
    map_size = N(size) + 1;
    if (! (map = make_bitmap(map_size)))
        fail("Insufficient memory");
    printf("bitmap: %lu bytes\n", bytes(SIZE(map)));

    // set composites
    sqrt_size = sqrt(dsize);
    t1=clock();
    for (i=5, incr=4, n=1; i<=sqrt_size; i+=(incr=6-incr), n++)
        if (BIT_OFF(map, n))            // bit clear -> prime
            for (j=i, jincr=incr; j<=size/i; j+=(jincr=6-jincr))
                SET_TRUE(map, N(i*j));  // bit set -> composite

    // count primes
    // printf("2 3 ");
    for (i=5, incr=4, n=1, primes=2; i<=size; i+=(incr=6-incr), n++)
        if (BIT_OFF(map, n))
        {
            primes++;
            // printf("%lu ", i);
        }

    runtime=(t2=clock())-t1;

    printf("\n%lu primes <= %lu\n", primes, size);
    printf("%.2f seconds\n", runtime/CLOCKS_PER_SEC);

    puts("Saving bitmap file PRIMES.DAT");
    f = fopen("primes.dat", "wb");
    fwrite(&size, sizeof(ULONG), 1, f);
    fwrite(map, 1, bytes(SIZE(map)), f);  // write out entire bitmap
    fclose(f);
}
```

The file BITMAP.H provides a BITMAP data type and a set of operations to set and test bits:

```
/* BITMAP.H */

typedef unsigned long ULONG;
typedef unsigned char BYTE;

#ifdef HUGE_MAP
// only required for 16-bit code
#define ALLOC            halloc
#define FREE             hfree
typedef BYTE huge *BITMAP;
#else
#define ALLOC            calloc
#define FREE             free
typedef BYTE *BITMAP;

#define SIZE(map)            (*((ULONG *) map))
#define index(c)             (((c) >> 3) + sizeof(ULONG))
#define mask(c)              (1 << ((c) & 0x07))
#define BIT_ON(map,c)        (map[index(c)] & mask(c))
#define BIT_OFF(map,c)       (! BIT_ON(map,c))
#define SET_TRUE(map,c)      map[index(c)] |= mask(c)
#define SET_FALSE(map,c)     map[index(c)] &= mask(c)
#define free_bitmap(map)     FREE(map)

void set_true(BITMAP map, ULONG c) { SET_TRUE(map,c); }

ULONG bytes(ULONG size)
{
    return (size >> 3) + ((size & 0x07) ? 1 : 0) + sizeof(ULONG);
}

BITMAP make_bitmap(ULONG size)
{
    BITMAP map;
    if (map = (BITMAP) ALLOC(bytes(size), 1))
        SIZE(map) = size;
    return map;
}
```

To compile this program for a 386 DOS extender using Watcom C 7.0/386, use the following DOS command line:

```
wcl386 -3r -mf -Oaxt -fpc sieve.c
```

The WCL386 driver program will first run the Watcom C compiler, WCC386.EXE (and the back-end code generator 386WCG.EXE), and will then invoke the linker, 386LINK.EXE (which you must purchase separately from Phar

Lap, and place somewhere on the DOS PATH). The resulting program, SIEVE.EXP, requires a 386 DOS extender. Phar Lap's DOS extender is contained in the loader program RUN386.EXE, and this can either be bound together with SIEVE.EXP to form SIEVE.EXE (assuming you have purchased a redistribution license from Phar Lap), or can be invoked from the DOS command line:

```
C:\EXTDOS>run386 sieve 30000000
Prime Number Theorem estimates: 1859537 primes <= 30000000
bitmap: 1250005 bytes
1857859 primes <= 30000000
Estimate off by 1678 (0.090319%)
233.000000 seconds
```

To compile the program using MetaWare High C 1.6, the command line is:

```
hc386 sieve.c
```

The HC386 driver first invokes the High C compiler, HCD386P.EXE, and then 386LINK. The Microsoft linker LINK.EXE cannot be used to produce 386 DOS extender applications. Note that, in contrast to the 286-based DOS extenders discussed in Chapter 4, no postprocessor (such as MAKEPM) is required.

Eclipse OS/386 can run Phar Lap executables. To distinguish these .EXP files from executables produced by another 32-bit linker, Lahey LINK-EM/32 (supplied with Lahey FORTRAN F77L-EM/32), Eclipse recommends renaming the file with a .PLX extension. There are two variants of Eclipse OS/386: a uniprocessor (UP) version, and a version using Eclipse's HummingBoard (HB) coprocessor. You can purchase a program to bind these OS/386 runtimes with the protected-mode executable to form a stand-alone DOS executable.

Now, this program can also be compiled with any other DOS C compiler. For example, a real-mode, large-model Microsoft C 5.1 version can be compiled and linked using the following DOS command line:

```
cl -AL -Ox sieve.c
```

But the resulting program has a fundamental limitation in real-mode MS-DOS: the 64K segment limit means that the bitmap must be less than 64K; this in turn means that, at most, this real-mode SIEVE.EXE can be used to find p(1,600,000).

In order to work around this limitation in real-mode MS-DOS, the program can be recompiled to use a "huge" pointer for the bitmap:

```
cl -AL -DHUGE_MAP -Ox sieve.c
```

But this only provides the program with an amount of memory less than 640K. Furthermore, huge pointers, while largely transparent to the programmer, impose a penalty in execution time.

It is interesting to note that, in order to take advantage of the larger address space available under one of the 286-based protected-mode DOS extenders discussed in the preceding chapter, the program still requires huge pointers. That is the only way, for example, to apply the benefits of DOS/16M or OS/286 to this program. One of the key differences between 286- and 386-based DOS extenders is that, while both break the 640K DOS barrier, only 386-based extenders also break the 64K segment barrier.

Since SIEVE.C can be compiled and linked for so many other environments, what makes it a 386 DOS extender program? In addition to removing space limitations, compiling this as a 32-bit application also produces an enormous jump in performance. Table 5-1 shows execution times for different versions of the SIEVE program, running on a 16 MHz Compaq 386 with 2 megabytes of memory:

Table 5-1: Execution times for a bitmap sieve.

			Seconds			
x	p(x)	Bitmap Size	MSC51 Large	MSC51 Huge	DOS/16M Huge	High C-386
100,000	9,592	4K	2	2	2	<1
1,000,000	78,498	41K	21	21	23	7
10,000,000	664,579	416K	N/A*	237	270	76
30,000,000	1,857,859	1.2M	N/A	N/A	827	236

*N/A—Insufficient memory

This comparison shows that, with identical source code and hardware, the 32-bit sieve runs more than three times faster than the 16-bit sieve. While this program is atypical in that it performs no I/O, its extensive manipulation of four-byte pointers and large data arrays are typical of most programs that one would consider porting to the 386.

Since all these programs were run on the same 386 machine, this test underlines the fact that running a program on a 386 does not make it a 386 program. To make good use of the 386 machine sitting on your desk, you need 32-bit software.

In order to remove as many restrictions as possible when running this code with 16-bit instructions, all indices were made unsigned longs (ULONG), and the `printf()` `"%lu"` mask was used. But in 32-bit code, a plain `unsigned int` would work equally well, as would the `printf()` `"%u"` mask, since an `int` is the same as a long in 32-bit C. sizeof(int) and sizeof(unsigned) are each 4, not 2. Likewise, sizeof(void *) is 4. (Note that the DOS-specific construct sizeof(void near *) is also 4, and that sizeof(void far *) is 6.)

Likewise, the all-important ANSI C identifier, size_t, which is the unsigned type of the result of the sizeof() operator and the type used by function parameters that accept the size of an object, is a four-byte quantity.

That has many ramifications for programming in 32-bit C. C standard library functions such as malloc(), fwrite(), and strncpy() all take size_t parameters, and strlen() returns a size_t. These standard library functions deal in quantities between 0 and UINT_MAX. In the 16-bit code generated by MS-DOS compilers such as Microsoft C, UINT_MAX is 0xFFFF (65,535), yielding the familiar 64K limit on PC array lengths, string lengths, and malloc blocks. But in 32-bit code, UINT_MAX is 0xFFFFFFFF, or 4,294,967,295: the magical upper "limit" of four gigabytes. In the native mode of the 386, this is the upper bound on array lengths, string lengths, and malloc() blocks: hardly a limit at all.

Using the SIEVE program to build a 1.2 megabyte bitmap to represent all prime numbers <= 30,000,000, we could save this entire bitmap to disk in one call to fwrite():

```
FILE *f = fopen("primes.dat", "wb");
fwrite(&size, sizeof(ULONG), 1, f);
fwrite(map, 1, bytes(SIZE(map)), f);   // write out entire bitmap
fclose(f);
```

This code would not work reliably when using an MS-DOS compiler such as Microsoft C or Turbo C. The third parameter to fwrite() is a size_t num_items, yet we are passing in an unsigned long; in addition, huge pointers cannot be reliably passed to standard-library functions. This is an example of how 16-bit mode forces the programmer to remember low-level aspects of the machine architecture. In contrast, 32-bit mode allows a far more "forgetful" style of programming, in which many more things work the way you wish they worked: passing a 1.2-megabyte buffer to fwrite() works just fine. "Normal" objects in 32-bit programming are true huge objects, without any of the limitations of what Microsoft calls huge objects.

How does this call to fwrite() actually work in a 32-bit DOS extender? Under MS-DOS, the C function fwrite() must eventually call Int 21H AH=40H (Write File or Device). A 32-bit DOS extender supports the Int 21H interface, even for objects that MS-DOS can't handle. We saw in Chapter 4 how a 16-bit DOS extender skillfully creates the illusion of an MS-DOS that can handle objects in extended memory. A 32-bit DOS extender must support not only this fiction, but also the fiction of an MS-DOS that can handle objects whose size is greater than 64K. Standard references to the MS-DOS programmer's interface carry the following description for the DOS `write` function:

```
Int 21H Function 40H
Write File or Device
BX = handle
CX = number of bytes to write
DS:DX = segment:offset of buffer area
```

In a subtle but important difference, the manuals for Phar Lap 386 | DOS-Extender, Eclipse OS/386, and IGC XAM show the following description:

```
Int 21H AH=40H
Write File or Device
BX = handle
ECX = number of bytes to write
DS:EDX = segment:offset of buffer area
```

The mention of the 32-bit ECX and EDX registers instead of the 16-bit CX and DX registers is crucial. In its underlying implementation, this eventually calls the "real" `Int 21H` Function `40H`, and so breaks up large requests into a series of smaller requests, and moves data from extended memory to conventional memory, But this is all transparent to the programmer, particularly the programmer using the standard library functions in a high-level language.

As noted later on, though, file I/O might be slower under a DOS extender than in real mode. In order to correct this, you might need to make sure that the DOS extender doesn't break your large `fwrite()` call into many tiny DOS calls. The C function `setvbuf()` is useful here, as are the Phar Lap command-line switches -MINIBUF and -MAXIBUF, which control the size of the low-memory data buffer used for DOS function calls. A program like SIEVE.EXP, which writes a large amount of data at one time, should allocate a large I/O buffer:

```
run386 -minibuf 32 -maxibuf 48 sieve 30000000
```

On occasion, you may have to be aware of small differences between the interface provided by a 32-bit DOS extender and that provided by MS-DOS, or between the different DOS extenders. The best example is `Int 21H AH=48H` (Allocate Memory Block). Real-mode MS-DOS expects in BX the number of 16-byte paragraphs of memory needed. (Since BX can hold values up to 65,535, this means that `Int 21H AH=48H` can be used to allocate 16 * 65,535 bytes at once, which is the basis for real-mode huge pointers.) Eclipse OS/386 mimics the DOS interface, expecting in EBX the number of paragraphs needed, but Phar Lap 386 | DOS-Extender instead expects in EBX the number of 4K pages an application needs! Code generated by a compiler should detect which environment it is running under and pass the proper parameters.

Adding 80386 Bit Test Instructions

Once we've decided to make SIEVE.C a 32-bit program, there are further improvements we can make. For example, the bitmap operations in SIEVE.C are typical of testing, setting, and resetting bits in C. To set a bit (turn it on), which the sieve program does a lot, you need:

```
#define index(bit)          (bit >> 3)
#define mask(bit)           (1 << (bit & 7))
#define SET_TRUE(map,c)     map[index(c)] |= mask(c)
```

C compilers such as Watcom 386 and MetaWare High C take `SET_TRUE (map,c)` and have to turn it into:

```
mov eax, _map
mov edx, _c
mov ebx, edx
shr ebx, 3
mov cl, dl
and cl, 7
mov dl, 1
shl dl, cl
or  4[ebx+eax], dl
```

This is better than 16-bit code, but is still not a very efficient way to operate on a bitmap.

As you may recall from our earlier discussion of the 386 instruction set, the Intel 80386, like the Motorola 680x0 family, has a set of bit test instructions. These are perfect for graphics, for implementing large sets, for semaphores, or for manipulating any sort of bitmap. Not exactly RISC! BT does a bit test, BTS sets a bit true, BTR resets a bit to false, BSF scans for the first set bit, and so on. The call `SET_TRUE(map, c)` could be implemented with:

```
mov esi, _map
mov eax, _c
bts [esi], eax
```

Such a CISC instruction takes longer to execute than a simple instruction like NOW or AND. But it's faster than the MOV/SHR/AND/SHL/OR series shown earlier, and certainly takes less room.

The bit test instructions also work in 16-bit code on a 386. But in 32-bit mode, since a 4-byte register can be used to hold the offset into the bitmap, the instructions handle maps with up to 4 gigabits (536 megabytes).

This is all well and good, but how can we get the compiler to use the 386 bit test instructions? If we were to code these as functions in a separate assembler module, any performance gain from using the bit test instructions would probably be lost in function-call overhead. However, many 386 C compilers come with a facility for in-line assembler. For example, the Watcom C `#pragma aux` facility can be used to describe a symbol's attributes, such as how a function receives it parameters, how it returns a value, and what registers its modifies. This makes it easy to tell the compiler how to generate code for a particular symbol. The following three lines tell the Watcom compiler to use the BTS instruction whenever it sees a call to `set_true()`:

```
void set_true(BITMAP map, ULONG bit);
#define BTS_ESI_EAX      0x0F 0xAB 0x06
#pragma aux set_true =   BTS_ESI_EAX     parm [esi] [eax] ;
```

This tells Watcom 386 that the block of code named `set_true` takes its parameters in ESI and EAX, but does not create an actual function `set_true()`. Instead, the call `set_true(map, c)` will now generate the following code:

```
mov esi, _map
mov eax, _c
bts [esi], eax
```

How much of a difference does this make to the performance of the sieve program? Whereas the Watcom 386 version took 74 seconds to find p(10,000,000) using standard C bit operations, a version that uses the 386 bit instructions takes 67 seconds: another 10 percent shaved off a program that was already three times faster than its 16-bit equivalent. A 386 replacement for BITMAP.H follows:

```
/* BTMAP386.H */

#if !defined(__WATCOMC__) || !defined(__386__)
#error BTMAP386.H requires Watcom C 7.0/386
#endif

typedef unsigned long ULONG;
typedef unsigned char BYTE;

#define ALLOC(x,y)        calloc(x,y)
#define FREE(x)           free(x)
typedef BYTE *BITMAP;
```

```
void set_false(BITMAP map, ULONG bit);
void set_true(BITMAP map, ULONG bit);
ULONG test_bit(BITMAP map, ULONG bit);
/* 386 bit test instructions */
#define BTR_ESI_EAX       0x0F 0xB3 0x06
#define BTS_ESI_EAX       0x0F 0xAB 0x06
#define BT_ESI_EAX        0x0F 0xA3 0x06

#define PUSHF             0x9C
#define POP_EAX           0x58
#define AND_EAX_1         0x25 0x01 0x00 0x00 0x00
#define MOV_EAX_CARRY     PUSHF POP_EAX AND_EAX_1

#pragma aux set_false =   BTR_ESI_EAX   parm [esi] [eax] ;

#pragma aux set_true =    BTS_ESI_EAX   parm [esi] [eax] ;

#pragma aux test_bit =    BT_ESI_EAX    MOV_EAX_CARRY \
                          parm [esi] [eax]  value [eax] ;

/* skip past 32-bit ULONG size header */
#define SIZE(map)         (*((ULONG *) map))
#define BIT_ON(map, i)    test_bit(map, (i)+32)
#define BIT_OFF(map, i)   (! BIT_ON(map, i))
#define SET_TRUE(map, i)  set_true(map, (i)+32)
#define SET_FALSE(map, i) set_false(map, (i)+32)
#define free_bitmap(map)  FREE(map)

// ... identical to tail of BITMAP.H
```

Here, we used Watcom C 7.0/386 because it is the most convenient for this task. For large commercial applications, however, you may find that MetaWare High C 386 is a more appropriate tool.

Virtual Memory

Another obvious 386 feature to use is virtual memory. Since the test machine we've been using has only two megabytes of memory, but a lot of free disk space, it would seem that this would be a perfect opportunity to try out the virtual memory option available with 386 DOS extenders. Unfortunately, though, a sieve is the worst possible test for virtual memory, since the program runs through the entire bitmap for every prime number found. While this implementation is fine if the entire bitmap is in memory, it would cause serious thrashing if any part of the bitmap was located on disk.

The following program, PRIMES.C, is a better demonstration of virtual memory under a 386 DOS extender. The program reads in the bitmap file PRIMES.DAT that was saved by the SIEVE program, and can be run on a 386 computer with less memory than the machine which created the PRIMES.DAT file. The program allows the user to type a '?' to query the prime-number bitmap, a 'V' to see virtual-memory statistics, or a 'Q' to quit. The following example not only shows the difference a little virtual memory can make, but also shows the mechanics of using Phar Lap's 386 | VMM:

```
C:\BOOK>run386 primes
Insufficient memory

C:\BOOK>run386 -vm \pharlap\vmmdrv primes
;;; A LOT OF DISK ACTIVITY ;;;
> ? 99998000099
9998000099 is not prime
Prime factors: 99989 99991
> v
VM active for 36 seconds
Page faults: 248
Pages written to swap file: 245
Reclaimed pages: 105
Application physical pages: 175
Application VM pages: 316
Swap file pages: 146
> ? 1000000000061
1000000000061 is prime
> q
```

When we tried to run PRIMES without benefit of virtual memory, the program complained and exited back to DOS. But with the virtual-memory manager, the program's call to `calloc()` succeeds. 386 | VMM is enabled by using the -vm flag on the DOS command line to RUN386. In the distribution version of an application, the VM driver would be bound together with the DOS extender and the application itself into a single .EXE file, and so would be invisible to the user. For example, Mathematica and UR/Forth both have 386 | VMM built into their executables.

In this session, the PRIMES program allocated 316 4K pages of memory, of which only 175 were located in physical memory. Thus, 141 pages of memory were located on disk in a swap file. The application must make a special system call to 386 | VMM in order to find these statistics, since in normal operation VM is invisible to the programmer. In the following source code for PRIMES.C, note that we allocate memory for the bitmap using the same `make_bitmap()` function used in SIEVE.C; this function in turn calls `calloc()`, which succeeds even though there isn't adequate physical memory to satisfy the request. There is a strong resemblance between VM and a government's ability to freely print paper money! Of course, here too there is no such thing as a free lunch, and VM opens the possibility of slower execution time than code using only physical memory.

In the following code, note that `Int 21H AX=2520H` is used to get VM statistics. As will be explained later, the interface to Phar Lap's API replaces MS-DOS's `Int 21H AH=25H`. Both MetaWare High C-386 and Watcom C/386 support a 32-bit extended version of the Microsoft C `intdos()` and `int86()` functions for invoking software interrupts:

```
/*
PRIMES.C
Author: Andrew Schulman, February 1990
*/

#include <stdlib.h>
#include <stdio.h>
#include <float.h>
#include <math.h>
#include <limits.h>
#include <dos.h>

typedef enum { FALSE, TRUE } BOOL;

void fail(char *s) { puts(s); exit(1); }

#include "bitmap.h"

#define N(x)               ((x) / 3)  // exclude multiples of 2 and 3

// don't test double for equality: DBL_EPSILON in <float.h>
#define EQ(x,y)            (((x) - (y)) < DBL_EPSILON)

BOOL is_prime(double x);
void prime_factors(double x);
void vm_stats(void);
void help(void);

BITMAP map;
ULONG size;
```

```
main()
{
    char buf[80], *s=buf;
    double d;
    FILE *f;
    ULONG map_size;

    if (! (f = fopen("primes.dat", "rb")))
        fail("requires PRIMES.DAT");
    fread(&size, sizeof(ULONG), 1, f);
    fread(&map_size, sizeof(ULONG), 1, f);
    if (! (map = make_bitmap(size)))
        fail("Insufficient memory");
    fseek(f, 4, SEEK_SET);
    fread(map, 1, bytes(map_size), f);
    for (;;)
    {
        printf("> ");
        gets(s);
        if (strlen(s+2) > DBL_DIG)
        {
            printf("Number too large\n");
            continue;
        }
        switch (toupper(*s))
        {
            case '?' :
                d = strtod(s+2,0);
                if (is_prime(d)) printf("%.0f is prime\n", d);
                else
                {
                    printf("%.0f is not prime\n", d);
                    prime_factors(d);
                }
                break;
            case 'Q' : fclose(f); exit(1);
            case 'V' : vm_stats(); break;
            default  : help(); break;
        }
    }
}

void help(void)
{
    puts("? [x] -- is x prime? if not, show prime factors");
    puts("Q -- quit");
    puts("V -- virtual memory stats");
}
```

```
BOOL is_prime(double x)
{
    ULONG i, n, lx, sqrt_x;
    int incr;

    if (x <= (double) size)
    {
        lx = x;
        if ((lx == 2) || (lx == 3)) return TRUE;
        if ((lx % 2) && (lx % 3) && BIT_OFF(map, N(lx))) return TRUE;
        else return FALSE;
    }
    else
    {
        if (EQ(fmod(x,2),0) || EQ(fmod(x,3),0)) return FALSE;
        sqrt_x = sqrt(x);
        for (i=5, incr=4, n=1; i<=sqrt_x; i+=(incr=6-incr), n++)
            if (BIT_OFF(map, n) && EQ(fmod(x,i),0)) return FALSE;
        // still here -- primes are residue
        return TRUE;
    }
}

void prime_factors(double x)
{
    ULONG i, n;
    int incr;
    printf("Prime factors: ");
    while (! EQ(x,1))
    {
        if (is_prime(x))     { printf("%.0f\n", x); return; }
        if (EQ(fmod(x,2),0)) { printf("2 "); x /= 2; }
        if (EQ(fmod(x,3),0)) { printf("3 "); x /= 3; }
        else for (i=5, incr=4, n=1; i <= x; i+=(incr=6-incr), n++)
            if (BIT_OFF(map, n) && EQ(fmod(x,i),0))
            {
                printf("%lu ", i);
                x /= i;
            }
    }
    printf("\n");
}

void vm_stats(void)
{
    ULONG buf[25];
    union REGS r;
#ifdef __WATCOMC__
    r.x.edx = buf;
    r.x.ebx = 0;         // don't reset VM stats
    r.x.eax = 0x2520;
#else
    r.x.dx = (unsigned) (void *) buf;
    r.x.bx = 0;
    r.x.ax = 0x2520;
#endif
    intdos(&r, &r);
    if (buf[0])          // VM is present
    {
        printf("VM active for %lu seconds\n", buf[11]);
        printf("Page faults: %lu\n", buf[12]);
        printf("Pages written to swap file: %lu\n", buf[13]);
        printf("Reclaimed pages: %lu\n", buf[14]);
        printf("Application physical pages: %lu\n", buf[5]);
        printf("Application VM pages: %lu\n", buf[15]);
        printf("Swap file pages: %lu\n", buf[16]);
    }
    else
        puts("VM not present");
}
```

Moving to 32-bit Programming

The preceding examples have shown that, if you are to have a PC background, then making the transition to 386 programming largely involves forgetting all the tricks you've learned to get around DOS and its real-mode limitations. Forget about dealing with objects in chunks smaller than 64K; forget about distinguishing between near and far memory; forget about distinguishing between different memory models; pretty much forget about segmentation. Under VM, forget about available memory (but you had better be aware of disk space, and of the cost of using it!).

You're likely to use a high-level language for the bulk of your development efforts. C is a popular development tool; from the outset of 386 DOS extender development, 386 C compilers have been readily available. MetaWare High C was the first available C compiler targeted for 386 DOS extenders; the High C 386 start-up code detects all three DOS extender runtimes and responds accordingly. Therefore, MetaWare High C-386 can be used for development targeted to work with any of the DOS extenders.

Most development work is likely to depend on the high-level language that the developer or team feels most comfortable with. If the primary programming task is porting a large application from a workstation, minicomputer, or mainframe environment, it makes sense to choose the 386 compiler that most closely corresponds with the mainframe compiler. Note that most FORTRAN 386 compilers support some idiosyncratic mixture of VAX and IBM mainframe anachronisms. Several 386 compiler vendors are from the UNIX marketplace, so their products are likely to support UNIX nuances and anomalies. If the application is being moved from the DOS world, better DOS compatible libraries are generally available from vendors moving up from the DOS marketplace. Those unusual programs to be written from scratch allow the most flexibility in choosing 386 development tools.

One major consideration in developing with C on the 80386 is that the widths of various data types are different from the widths of equivalent DOS counterparts. As noted earlier, on the 386, the `int` (integer) data type is now a full 32 bits wide, comparable to the `long` data type under 8086 DOS C compilers. A side benefit is that code from the UNIX and minicomputer world which assumed that an `int` was 32 bits wide will port quite easily to the 386.

If the major part of a programming project is in a high-level language (Ada, C, Pascal, or Fortran, for example), porting to the 386 DOS extenders can be a relatively painless task. High-level language compiler vendors have complied with protected-mode restrictions by modifying their libraries for protected mode, and can shield the programmer from much of the changes wrought by memory and hardware protection. Such problems are more likely to arise in small sections of a large application that have traditionally been hand-coded in assembly language. Assembly language may have been used for the following reasons:

- faster execution speed
- smaller memory space requirements
- the need to communicate directly with video or other hardware
- the need to interface with existing real-mode libraries.

A reasonable strategy is to minimize assembly language usage and focus on the best algorithms at the compiler language level. A parallel routine in C that operates slowly can always be replaced at a later phase with a faster hand-coded 80386 assembly language function.

Although useful and sometimes necessary, interfacing with real-mode libraries will commonly create more problems than are solved. Although real procedure call (RPC—analogous to, but not to be confused with, remote procedure call in networking) and signal mechanisms are available with DOS extenders (the OS/386 RPC mechanism is quite elegant), a steep learning curve is involved in coming up to speed with these special DOS extender features. If source code for real-mode libraries is available (whether in assembly language or a high-level language), it is almost always preferable to revise and recompile or reassemble the source code expressly for the 386/486.

When programming in 386/486 protected mode, the developer must discipline the casual programming style typical of programming under DOS. Access to physical memory is no longer direct. Memory protection is enabled, and segment registers must be used with more care. Segment registers can no longer be used for arithmetic. Any value loaded into a segment register must now be a legitimate selector. As noted in Chapter 1, a selector represents an index to an entry in a local or global descriptor table. These descriptors hold information about the segment type, length, privilege level, and base address. Without paging, the base address is the actual physical address in memory. With paging enabled, yet another level of logical-to-physical address translation takes place. In any event, actual addresses are no longer directly accessible to a program.

In addition, a code segment (selector) is marked read-only/executable and usually can't be written to. Actually, 386 DOS extenders support a data selector alias for the code segment that does allow such programming practices.

A selector will also have a limit (size), which, if exceeded, can cause a memory protection violation. Even the stack can cause a memory protection violation if not enough stack space has been allocated and marked in the descriptor table.

This opens the question of how to handle memory-mapped video and graphics under a 386 DOS extender. For speed, high-end applications often choose to write directly to video memory, bypassing the ROM and video ROM BIOS. Both text and graphics modes are often handled in this way. Because of memory protection issues in protected mode, video memory cannot be accessed at an absolute physical address, as it is under DOS. Under DOS extenders, a selector that points to the actual block of video or graphics memory must be used. All of the DOS extenders provide some built-in mechanism to readily address video memory. This mechanism is usually a hard-wired selector pointing to a chunk of video RAM. Also, extended functions provided by the DOS extenders allow allocating a memory segment for any block of physical memory. These functions provide a scheme to handle most atypical video and graphic card options.

Access to the interrupt vector tables and hardware can be more problematic than in simple DOS programming. The interrupt vector table is internal to the DOS extender. To install a replacement interrupt handler requires using a 386

DOS extender function. There can be two interrupt handlers for each hardware interrupt:

- a real-mode handler that gains control when DOS is executing in lower memory
- a protected-mode handler that gains control when the program is operating in protected mode and the DOS extender is active.

DOS extenders all support some way to pass the real-mode interrupt up to a protected-mode interrupt handler and some way to pass interrupts from protected mode down to a real-mode handler. DOS extenders can also provide features to install dual interrupt handlers, which operate in both situations. Dual interrupt handlers provide the best interrupt response, because the DOS extender does not have to perform a switch from real to protected mode in order to process an event. The most common use for dual interrupt handlers would be a serial device that generated many interrupts (graphics tablet, plotter, and high-speed modem. A dual interrupt handler could be set up to share a common data buffer in low memory to maximize response time. On fast 80386 machines, dual interrupt handlers are less necessary.

Tools for 386 DOS Extender Programming

All three 386 DOS extenders include a debugger, utility programs, and examples of their use to create and execute protected-mode 386 programs. All three emulate DOS and the BIOS to a substantial degree. For most conventions, the Eclipse and Phar Lap extenders are similar enough to be used somewhat interchangeably. The three runtimes support four different executable file formats (Phar Lap supports both an old and a new EXP executable). From the programmer's perspective, it would be preferable if the three runtimes were closer together in their DOS and BIOS emulation and EXE file formats, and in the way they handle other hardware issues (the Weitek chip, for example).

Choosing a 386 DOS Extender

Most programmers would rather choose 80386 compilers and development tools without being forced to make a choice of development and runtime environments at the same time. Unfortunately, with the exception of a few compilers such as MetaWare High C-386, choosing a 386 compiler locks a programmer into its associated DOS extender and debugger.

Let's take a typical application and divide the time spent on various tasks. You might spend 20 percent on planning and design, 40 percent on programming, and 40 percent on debugging. When you decide on a runtime environment, you are also choosing a debugger and technical support. Because of software differences among them, a debugger from one vendor will not work in another 386 environment.

There is one disadvantage to the near-absence of segmentation in 32-bit protected-mode; it makes debugging fairly difficult. While page-level protection is still available when using the 386 as a linear address space, it does not provide nearly as adequate protection as segment-based protection in 16-bit protected mode. While the 386 provides a new set of registers for debugging, to date actual debugging facilities for 386-based DOS extenders are more primitive than under real-mode DOS, 286-based extended DOS, or OS/2. One of the benefits of 16-bit protected mode's extensive use of segmentation is the support it provides for debugging: many software developers might use OS/2 and 286-based DOS extenders for this, if for nothing else.

Phar Lap 386|DOS Extender

Phar Lap markets 386 | DOS Extender (RUN386), a 386 assembler, linker, librarian, and debugger package (386 | ASM/LINK), a symbolic debugger (386 | DEBUG), and tools for embedded applications (LinkLoc). Phar Lap has been the traditional supplier for the assembler and linker used for 386 development under DOS. Phar Lap defined the 386 object module (EASY-OMF) most commonly used under DOS. EASY-OMF is an extension of the Intel OMF-86 object module, in which some fields are extended to 32 bits for the 80386. An EASY-OMF object module is denoted by a comment record at the start of the file containing "80386." With the exception of Lahey FORTRAN F77/L32, all 386 compilers can generate Phar Lap 386 object files and are supplied with Phar Lap compatible libraries. (The Lahey compiler uses the Microsoft 32-bit object module format, and requires the Lahey linker, L32.)

Phar Lap's assembler and linker are provided in two forms under DOS: a real-mode version, and a protected-mode version that runs under DOS on an 80386 with more than 1 megabyte of extended memory. The protected-mode version operates much faster but is otherwise equivalent to the real-mode version. FASTLINK, the protected-mode version of 386LINK, can link larger programs. 386 | DEBUG is very similar to Microsoft's SYMDEB and supports symbolic debugging at the assembly language level.

RUN386, the 386 DOS extender, supports a number of switches and options, and automatically senses the presence of other programs using extended memory (RAMdisks, EMS emulators, and so on). RUN386 supports the Virtual Control Program Interface (VCPI) drafted with Quarterdeck Systems to allow multiple 386 programs to cooperate, averting the chaos that characterizes TSRs under DOS. RUN386 supports calls from protected mode to real mode. Phar Lap was the first vendor to support virtual memory for 386 applications, and, as noted earlier, a number of commercial products already incorporate the Phar Lap Virtual Memory Manager (386 | VMM). Another interesting aspect of 386 | DOS-Extender is its support for protected-mode TSRs: when a protected-mode program makes a TSR system call (e.g., `Int 21H AH=31H`), both the protected-mode program and RUN386 stay resident in memory.

Phar Lap developed and supports two different EXP file formats:

- the original EXecutable Protected mode (EXP)
- a P3 format EXP, which now supports a packed mode that creates smaller executable file sizes.

Although RUN386 can load a Relocatable EXecutable (IGC's REX) file (Phar Lap wrote the assembler/linker originally used by all three vendors), it cannot, in general, execute REX programs because of differences in the Phar Lap and IGC runtime environments.

The Phar Lap runtime is a flat memory model with several hardwired segment selectors for memory mapping. For example, a program CS is set to `0CH`, and DS and ES are set to `14H`—all pointing to the same block of physical memory. There are also hardwired selectors for the video refresh buffer, the lower 1 megabyte of memory, and the program's environment block and program segment prefix (PSP). These are listed in Table 5-5 (Phar Lap and Eclipse LDT hardwired selectors), later in this chapter.

Normally, both RUN386 and your application run at ring 0 (most privileged). This allows complete machine control from within your application. RUN386 can turn paging on or off. With paging enabled, RUN386 can also use memory below 640K for 386 applications. The Phar Lap DOS extender uses about 100K of low memory aside from DOS, leaving about 500K of low memory, plus any extended memory above 1 megabyte available for protected-mode use. This DOS extender makes the largest quantity of memory available for 386 programs. This can be a consideration for machines with relatively low memory (2 megabytes).

DOS protected-mode function calls are reasonably similar to normal DOS calls, with a few exceptions. Of course, registers are 32-bits wide instead of 16 (EAX versus AX). DOS `GetVersion` `(AH=30H)` is used to return information about the runtime. The DOS `Get Vector (AH=35H)` and `Set Vector (AH=25H)` calls are not supported, but are replaced with DOS extender calls. FCB-type file I/O calls are not supported. Memory management and EXEC calls are slightly different from their DOS counterparts. The Weitek math chip is supported by mapping it into a 64K block of memory pointed to by segment register FS.

Phar Lap provides a set of system calls via `Int 21 AH=25H`. Since this MS-DOS function to set interrupt vectors was likely to be changed anyway for 32-bit protected mode, Phar Lap chose to use this as the interface to all Phar Lap system services. These services include support for protected- and real-mode interrupt handling, memory management, intermode communication (calling real-mode procedures from protected mode), and virtual memory. A service is chosen with a function number in the AL register. Below is a listing of the various functions:

Interrupt Handling

AH=25H AL=02H	Get protected-mode interrupt vector
AH=25H AL=03H	Get real-mode interrupt vector
AH=25H AL=04H	Set protected-mode interrupt vector
AH=25H AL=05H	Set real-mode interrupt vector
AH=25H AL=06H	Set interrupt to always gain control in protected mode
AH=25H AL=07H	Set real- and protected-mode interrupt vectors
AH=25H AL=0CH	Get hardware interrupt vectors (IRQ0-15)

Memory Management

AH=25H AL=08H	Get segment linear base address
AH=25H AL=09H	Convert linear to physical address
AH=25H AL=0AH	Map physical memory to end of segment
AH=25H AL=13H	Alias segment descriptor
AH=25H AL=14H	Change segment access rights or USE16/USE32 flag
AH=25H AL=15H	Get segment access rights and USE16/USE32 flag
AH=25H AL=16H	Free all memory owned by LDT
AH=25H AL=18H	Specify handler for moved segments

Real-mode Communications

AH=25H AL=0DH	Get information for real-mode function call
AH=25H AL=0EH	Call real-mode procedure, stack-based

AH=25H AL=0FH	Convert protected-mode address to MS-DOS
AH=25H AL=10H	Call real-mode procedure, register-based
AH=25H AL=11H	Invoke real-mode software interrupt
AH=25H AL=17H	Get information on DOS data buffer

Virtual Memory (VMM)

AH=25H AL=19H	Get additional memory error information
AH=25H AL=1AH	Lock pages in memory
AH=25H AL=1BH	Unlock pages
AH=25H AL=1DH	Read page-table entry
AH=25H AL=1EH	Write page-table entry
AH=25H AL=1FH	Exchange two page-table entries
AH=25H AL=20H	Get virtual memory statistics
AH=25H AL=21H	Limit program's extended memory usage
AH=25H AL=22H	Specify alternate page-fault handler
AH=25H AL=23H	Specify out-of-swap-space handler
AH=25H AL=24H	Install page-replacement handlers
AH=25H AL=25H	Limit program's conventional memory usage

Miscellaneous

AH=25H AL=01H	Reset 386 I DOS-Extender data structures
AH=25H AL=12H	Load program for debugging
AH=25H AL=26H	Get configuration information
AH=25H AL=C3H	Execute program
AH=25H AL=C0H	Allocate MS-DOS memory block
AH=25H AL=C1H	Release MS-DOS memory block
AH=25H AL=C2H	Modify MS-DOS memory block
AH=30H EBX="PHAR"	Get 386 I DOS-Extender version

The majority of 386 compilers run under, and generate code to operate using Phar Lap's DOS Extender. These include products from MetaWare, SVS, Micro-Way, Alsys, and Watcom.

Eclipse Computer Solutions OS/386

Eclipse Computer Solutions markets two DOS extenders: OS/386, designed for 386 machines and their Hummingboard 80386/80387 coprocessor boards, and OS/286, a 16-bit DOS extender, already discussed in Chapter 4.

Eclipse's Developer's Kit includes the OS/386 kernel, a symbolic debugger, and several utility programs. Eclipse's OS/286 supports a multiple segmented memory model on the 286, allowing applications to break the 640K memory barrier under DOS. The same memory map is supported on the OS/386 runtime, along with the more common flat, unsegmented memory model. For development purposes, the kernel is installed as a TSR (which can be removed from memory when not needed). This speeds the loading of protected-mode programs during both development and testing. The kernel runs at ring 0 (most privileged), while your protected-mode program runs at ring 3 (least privileged). The I/O privilege level (IOPL) for applications is set at 3 so that your application can input and output to hardware ports.

The OS/386 DOS extender's support of VCPI allows programs to run under Quarterdeck's DESQview 386, which provides multitasking capabilities. VCPI adherence also provides for compatibility with other conforming DOS extender applications. The resident OS/386 kernel can be configured to use only about 60K of the lower 640K DOS memory space, freeing more memory for other real-mode applications and TSRs. Memory management features include multiple heaps, automatic compaction, and control over where the protected-mode component of the kernel is loaded—below or above the 1-megabyte boundary. Memory management service functions allow access to page tables, selection of low or high memory heaps, and control over compaction. Interrupt handlers can be easily chained to real-mode handlers or shared between parent and child tasks. A supplied setup program tunes the OS/386 DOS extender runtime for optimum performance on 386 machines.

Eclipse's protected-mode programs use the file extension .EXP (EXecutable Protected mode) for both 16-bit and 32-bit programs. Unfortunately, this is the name Phar Lap adopted for their default executable files even though Eclipse's and Phar Lap's files have different structures. Eclipse can run the Phar Lap 32-bit files, using either the .PLX (Phar Lap eXtended) or .EXP extension. A strength of the Eclipse product is its support for both 80286 and 80386/486 protected-mode DOS.

Among the three runtime environments, OS/386 offers the closest emulation of DOS and BIOS. The OS/386 manual describes both compatible and slightly incompatible DOS calls supported. Primary variances from DOS are with FCB I/O calls (records are limited to 16K), DOS memory allocation, and EXEC calls. Otherwise, all DOS calls are fully supported, except that 32-bit registers are used.

A number of extended DOS calls are supported for calling real-mode procedures, setting arbitrary interrupt vectors, creating code and data segments, getting segment information, and doing block transfers to low memory. These services are invoked with `Int 21H` with `AH` ranging from `E0H` to `EDH`, and are identical to those listed in the section of Chapter 4 on *OS/286 and the Two-Machine Model*—an indication of the strong ties between OS/286 and OS/386.

As an option, Eclipse provides a demand-paged virtual-memory version of the OS/386 TSR. This is similar to the Phar Lap virtual-memory manager discussed earlier. In addition to its normal transparent operation, the Eclipse virtual-memory option adds the following extended functions to the OS/386 `Int 21H` programming interface:

■ `AH=EBH AL=00H`	Get a page table entry (PTE) by linear address
■ `AH=EBH AL=02H`	Get a page table entry (PTE) by 16-bit segment:offset
■ `AH=EBH AL=03H`	Free mapped pages
■ `AH=EBH AL=04H`	Get a page table entry (PTE) by 32-bit segment:offset
■ `AH=EBH AL=05H`	Map pages
■ `AH=EBH AL=06H`	Lock pages in memory
■ `AH=EBH AL=07H`	Unlock pages.

The Eclipse debugger, Command Processor (CP), is a command shell that can execute DOS-like built-in commands and batch files, as well as be used as an assembly language symbolic debugger. The shell includes a history command and a built-in command line editor using key bindings similar to the standard EMACS editor defaults. The shell has a macro processor that allows invoking a batch file of macros to make the debugger or shell look similar to SYMDEB, and other user interfaces are only a macro file away. A utility converts a link map or object file into a symbol file suitable for use with CP, providing symbolic debugging. During development, execution of a 386 program is preceded by `UP` (uniprocessor) or, if using Eclipse's 386 coprocessor, by `HB` (HummingBoard), on the DOS command line.

OS/386 supports a real procedure call (RPC) mechanism that facilitates communication with real-mode routines, such as graphics and communications, located in low memory. RPCs allow 386 applications to use the extensive real-mode libraries until 386 versions of the libraries are available. This OS/386 RPC mechanism allows RPCs to be written in either C or assembly language, and is discussed at length later in this chapter.

Currently, the 80386 compilers that support the OS/386 runtime and produce 32-bit code are MetaWare High C-386 and Professional Pascal-386, Watcom C/386, and Lahey FORTRAN F77/L32. The Lahey 32-bit linker and librarian are supplied for development along with several other utilities.

IGC X-AM Development Environment

IGC offers two families of 80386 system products—VM/386 (a multitasking control program similar to Microsoft Windows/386 and Quarterdeck DESQview) and X-AM, their 386 DOS extender. Although the IGC X-AM (eXtended Address Mode) runtime, called VM/RUN, doesn't implement virtual memory, the overall design is based on creating virtual machines with complete protection of the operating system from damage by an errant program.

The other DOS extenders run DOS in real mode and 386 applications in protected mode. VM/RUN puts the entire system in protected mode and runs the DOS kernel in lower memory in the virtual 8086 mode. VM/RUN provides a completely flat memory model for the system, with applications loaded at the 32-megabyte address. No hardwired segments are used. Monochrome screen video memory is mapped at offset B0000H, and BIOS data at 400H—their actual physical addresses. All of this is done with paging. Because paging cannot be turned off, the IGC runtime can exhibit an Intel bug in older 386 machines that occurs only when 387 instructions are executing and paging is enabled (see the later discussion under *Hardware Requirements*).

VM/RUN is the largest DOS extender (200K) and takes up the most memory. After it is loaded and has allocated buffers for I/O, EXECing other applications and TSRs, memory below 640K is pretty much used up. These memory protection features are justified for VM/386; the necessity for these features in the single-tasking X-AM system, however, is arguable. X-AM supports the Weitek math chip, memory-mapped up high in the 386 address space, at the same location UNIX vendors chose on the 386.

Of the three DOS extenders, VM/RUN provides the weakest emulation of DOS. VM/RUN doesn't support a number of system calls, or supports them differently than DOS does. On startup, X-AM passes a global data structure (GDA) with system information to your application. (MetaWare High C-386 1.6 provides an include file, GDA.H, for accessing this structure.)

X-AM requires a REX (Relocatable EXecutable) file extension for 386 programs. The REX file can be generated by the Phar Lap 386 linker and includes relocation information. The runtime consists of four REX files and a main loader file (VMRUN.COM). The VM/RUN REX files must be available along the current path. For debugging, a special debug REX file must be substituted for one of these standard X-AM REX runtime files. The main COM file is a program loader used to make profiles for the executable 386 program. Similar to a feature found in UNIX, the loader uses `arg[0]` (the name of the command invoked) under DOS 3.x to determine the REX file to load. A utility supplied with the runtime concatenates these files into a single executable for distribution. But during development this then requires an additional 110K loader file with the same name as your 386 REX application file. This makes more sense under UNIX, where multiple links (with different names) to the same physical file can exist. Under DOS during the development stage, each one of your REX programs carries this extra file around.

Under VM/RUN, the stack is completely protected and doesn't grow in size. You use a utility program to set the maximum stack size in executable programs. During development, the first execution of a program will typically crash with a stack protection fault until the stack size is increased. The X-AM assembly language debugger is the weakest of the three DOS extenders. VM/RUN does not currently incorporate the VCPI interface, nor recognize when other programs are using extended memory.

Some of the items in the IGC VM/RUN Global Data Area (GDA) may be useful to your program. These include the PSP address, the data transfer address (DTA), the application start address, pointers to interrupt tables, the code and data selector for the application, available high and low memory, stack parameters, and pointers to other data structures. These other data structures in turn contain pointers to the GDT, IDT, page directory tables, an asynchronous terminal profile block for the COM ports, and a variety of internal data fields and working areas used by VM/RUN itself.

An entry in the GDA (GDA_SERV) points to a routine that provides a variety of extended services to VM/RUN. These extended functions can also be accessed from an application. X-AM does not use software interrupts for this interface. Instead, the AH register is loaded with `A0H`, AL is loaded with a subfunction number, and then EAX is loaded from the 386's CR0 register. Rather than overwrite the contents of EAX, `MOV EAX,CR0` is a privileged instruction in Ring 3 protected mode and in Virtual 8086 mode, and causes a trap which allows X-AM to gain control. For example, to get the address of the GDA, an application running under XAM would do the following:

```
gdaptr  dd  0
;...
```

```
        mov eax, 0A007h        ; subfunction 7: get GDA address
        mov eax, cr0           ; invoke extended function
        mov gdaptr, edx        ; the GDA pointer is returned in EDX
```

A list of these extended functions is shown below:

Interrupt Handling

AH=A0H AL=04H	Issue a soft IRET from Virtual 8086 mode

Memory Management

AH=A0H AL=01H	Move memory
AH=A0H AL=05H	Load a real address from a virtual address
AH=A0H AL=06H	Relocate a memory block

Real-mode Communications

AH=A0H AL=08H	Call a 386 process from a Virtual 86 process
AH=A0H AL=09H	Restart a 386 process from a Virtual 86 process
AH=A0H AL=0AH	Call a virtual 86 routine from a 386 process

Miscellaneous

AH=A0H AL=00H	Initialize GDA from Virtual 86
AH=A0H AL=02H	Transfer from Virtual 8086 to 386
AH=A0H AL=03H	Exit Virtual 86 and return to VM/RUN in real mode
AH=A0H AL=07H	Get GDA address

Limitations and Trade-offs

Each of the 386 DOS extenders has a certain amount of overhead associated with it. In terms of memory used by the DOS extender, the difference is about 500K from smallest memory needs (Phar Lap) to largest (IGC's X-AM). On 386 machines with limited installed memory (2 megabytes, for example), this difference can represent a sizable chunk of potential program and data space.

The DOS extenders are not ideal candidates for fast file I/O. Whenever a DOS call is made by a protected-mode application, the machine state is saved twice—once by the DOS extender and once by the underlying DOS system. Also, under normal circumstances, file I/O is handled by the real DOS and BIOS down low in physical memory, and block moved by the DOS extender to your program's disk buffers in high memory. This additional block move and save will impose a performance penalty on file I/O. Therefore, it is best to minimize DOS file calls and perform file I/O in large chunks if possible. These are good recommendations even in real-mode programs.

When programs operate in protected mode, memory protection has its benefits and costs. From a development standpoint, some programming errors that are often overlooked or missed when using DOS show themselves quite dramatically as memory protection violations. These include null pointer assignments, bad pointer values that exceed the limits of the data selector, and writing carelessly into code segments. Depending on the runtime, the DOS extender system may actually crash and reboot as a result of the processor exception. This is particularly true of stack violations. In any case, a crashed DOS extender makes for difficult debugging.

Another limit is that the debuggers supplied or available from the 386 DOS extender vendors are primitive by current PC programming standards. All three DOS extender debuggers are, at best, symbolic debuggers at the assembly language level. Some of the compiler vendors (SVS and Watcom) either bundle a high-level language debugger with their compiler or make one available as an optional product. These language debugging tools are certainly an improvement over stepping back in time to assembly language debugging.

Assembly Language

Assembly language tools for the 386 have traditionally been supplied by Phar Lap, as part of its "80386 Software Development Series." Its 386 | ASM/LINK package includes an assembler, a linker, a librarian, and a mini-debugger. The Phar Lap assembly language tools generate EASY-OMF object modules, which are supported by most 386 compiler vendors.

Microsoft MASM 5.0+ can also be used to assemble 80386 code, and generates a different (Microsoft extension) 386 object module format. Microsoft does not offer a linker or librarian to handle 386 object files. Eclipse bundles with OS/386 a 386 linker and librarian written by Lahey Computer Systems, which handles both Microsoft and Phar Lap 386 object modules. The Lahey librarian can also convert Phar Lap libraries to their own 386 Lahey library format. These tools are the same language utilities supplied with Lahey F77/LEM-32 (their 386 FORTRAN development system).

MetaWare bundles with their High C 386 and Professional Pascal 386 a useful binary dump utility (BD.EXE) that handles both Phar Lap and Microsoft 386 object modules, and that can convert from one object module format to the other.

High-Level Languages

In 1987, MetaWare High C-386 and Professional Pascal-386 were the first compilers available for the 386 DOS extender environment. In fact, 386 DOS extender development depended on the availability of MetaWare High C-386, which was used for generating parts of the runtimes, assemblers, linkers, librarians, and debuggers that make up these environments. Table 5-2 shows the wide range of high-level languages that are now available for developing 386 DOS extender applications.

Table 5-2: Programming languages for 386 DOS extenders.

Language	Vendor	Product
ADA:	Alsys	Alsys Ada 386
	RR Software	386/ADA
	Telesoft	Telesoft-Ada
APL:	STSC, Inc.	APL-PLUS II
	dyadic	dyalog APL/386
BASIC:	Language Processors, Inc.	LPI Basic
	Silicon Valley Software	SVS 386/BASIC PLUS
	TransEra Corp.	HTBasic
C:	MetaWare	High C-386
	MicroWay, Inc.	NDP C-386
	OASYS (Green Hills)	C-386
	Silicon Valley Software	SVS 386/C
	Watcom Systems, Inc.	Watcom C /386
C++:	INTEK	C++
	MicroWay, Inc.	NDP C++
COBOL:	Language Processors, Inc.	LPI Cobol
FORTH:	Laboratory Microsystems (LMI)	UR/FORTH
FORTRAN:	Lahey Computer Systems	F77L-EM/32
	Language Processors, Inc.	LPI FORTRAN
	MicroWay, Inc.	NDP FORTRAN-386
	OASYS (Green Hills)	FORTRAN-386
	OTG Systems, Inc.	FTN77/386
	Science Applications (SAIC)	SVS FORTRAN-386
	Silicon Valley Software	SVS 386/FORTRAN
LISP:	Symbolics, Inc.	CLOE-386

Language	Vendor	Product
PASCAL:	MetaWare	Professional Pascal-386
	MicroWay, Inc.	NDP Pascal-386
	OASYS (Green Hills)	Pascal-386
	Science Applications (SAIC)	SVS Pascal-386
	Silicon Valley Software	SVS 386/Pascal
PL/I:	Language Processors, Inc.	LPI PL/I
PROLOG:	Epsilon	MProlog
	Expert Systems Int'l Inc.	Prolog-2
SMALLTALK:	ParcPlace Systems	Smalltalk-80/386
SPITBOL:	Catspaw	Spitbol-68K/386

The 386 DOS Assembly Language Interface—How It Works

DOS and the PC ROM BIOS, combined, provide operating system services in five general areas:

- the file system
- I/O (keyboard, screen, printer, etc.)
- memory management
- processor management
- other information (clock, critical errors, etc.).

In addition, real-mode DOS can be bypassed if you install an interrupt handler to replace or enhance some DOS or BIOS facility.

A DOS extender can be perceived as a protected-mode version of DOS. Programs running under a DOS extender are provided services similar to those provided to a program running in real mode. From the perspective of the protected-mode application, the operating system looks like a 32-bit MS-DOS. The DOS extender, in turn, looks like an application to the actual DOS operating system located in low memory. 386 DOS extenders generally pass file management, I/O, and other information requests on to DOS. The results returned from DOS are passed back through the DOS extender to the application. Processor and memory management, however, except memory allocation in the lower 640K, are handled solely in protected mode by the DOS extender.

To use protected mode, software must first set up the prerequisite memory management tables, including segmentation and paging tables, if paging is enabled. The memory management tables are the global (GDT), the local (LDT), and the interrupt descriptor tables (IDT), discussed in Chapters 1 and 4.

In addition, some gateway or bridge to DOS must be constructed—this is the task of the 386 DOS extenders. These control programs set up the required memory management structures, including a gateway to DOS and other operating system and hardware services, load a 386 program into memory, and start it. In general, these runtime extenders set up an emulation of the DOS and BIOS calling conventions, using `INT` instructions. They intercept DOS calls, transform them for passing to the real DOS kernel that sits in low memory, pass the call to the real DOS and BIOS, and pass the returned information back to your 386 program. The subtle differences between one DOS extender and another lie in this emulation of DOS and BIOS that your application sees, and in special features—for instance, the DOS extender can allow a 386 protected-mode program to call an 8086 real-mode procedure (say, a graphics routine or a TSR) sitting down in low memory, or load other 80386 programs.

DOS extenders appear like DOS to a 386 application. If DOS or BIOS services need to be called directly from an application, the DOS or BIOS emulation provided by the DOS extender needs to be understood. DOS calls are still made by loading a function in register AH and executing an `Int 21H`. The primary difference is that the registers used are now a full 32 bits wide. The most common mistake is to forget that most pointers need to be a full 32 bits to point at an object.

The sample code below illustrates a simple assembly language function to emulate the UNIX clock routine. This function calls the DOS `GetTime` function and returns the number of 1/100 seconds since midnight. The assembly language code uses a few fast 80386 instructions for zero-extending a register (MOVZX) and small integer multiplies (`LEA`). This routine can be used with most 386 compilers that expect the results of a function in the EAX register. This particular function has been used to time the code generated by various compilers for speed of execution. The name might need to be changed depending on the compiler chosen: some C compilers emit C function symbols with a leading underscore; most Fortran compilers expect uppercase names without underscores.

```
; CLOCK.ASM
; A clock function for NDP C-386, LAHEY F77-EM/32, SVS 86/FORTRAN, etc.
; returns 1/100 seconds since midnight as a long.
; assemble with either Phar Lap 386|ASM or Microsoft ASM
; C calling convention
;
; extern long clock();
; long t;
; t = clock();
```

```
.386                          ; required to generate 32-bit code/data

; It is important when linking to F77L-EM/32 program units that data
; segments your assembly code uses have class name 'DATA' in order
; to link correctly. You must also use the GROUP directive to
; include the data in DGROUP. DS and ES are set to DGROUP by the
; FORTRAN code, and must be set that way on return. Your code segment
; must also be included in CGROUP and include the directive:
; ASSUME DS:DGROUP, CS:CGROUP

dataseg segment dword
dataseg ends
;
codeseg segment para public use32
assume cs:codeseg, ds:dataseg
;
public _clock

align 4                       ; align location counter

_clock proc near
   mov ah,2ch                 ; DOS Get Time function
   Int 21H                    ; CH=hour, CL=min, DH=sec DL=hundredths
; mov eax,60                  ; USING LEA INSTEAD OF MUL
; mul ch                      ; FOR FAST INTEGER MULTIPLY
   movzx eax,ch               ; start with hours
   lea eax,[eax+eax*4]        ; x 5
   lea eax,[eax*4]            ; x 20
   lea eax,[eax+eax*2]        ; x 60
;
   movzx ecx,cl               ; zero top of register
   add eax,ecx                ; now add in minutes
   lea eax,[eax+eax*4]        ; x 5 use some fast integer multiplies
   lea eax,[eax*4]            ; x 20
   lea eax,[eax+eax*2]        ; x 60
;
   movzx ecx,dh
   add eax,ecx                ; add in seconds
   lea eax,[eax+eax*4]        ; x 5 use some fast integer multiplies
   lea eax,[eax*4]            ; x 20
   lea eax,[eax+eax*4]        ; x 100
   movzx ecx,dl
   add eax,ecx                ; now add in hundredths of seconds
;
   ret
_clock endp

codeseg ends
end
```

Special DOS Extender Features

A number of special features are provided by the various 386 DOS extenders, including the following:

- writing directly to video memory
- writing into code segments
- installing interrupt handlers (real and protected mode)
- real-mode procedure calls (RPCs)
- virtual memory and page locking extensions.

In general, these features or options are implemented differently depending on the DOS extender runtime.

Writing Directly to Video Memory

All three DOS extenders discussed in this chapter provide the capability to write directly to memory-mapped video for fast screen output. Both Phar Lap and Eclipse provide hardwired selectors that point to the default video RAM (B000H or B800H are typical). IGC uses the linear model so that video RAM is addressed at an offset corresponding to its physical address (B0000H or B8000H). Both Phar Lap and Eclipse, however, support extended DOS function calls to map any physical address to a selector (segment). This mechanism is the most general one and will handle almost any memory-mapped device (standard video, high resolution TMS34010 graphics cards, network cards, and SCSI adapters).

When a protected-mode program is loaded into memory, the Phar Lap and Eclipse DOS extenders set up a number of hardwired segments (see Table 5-3). These selector values are different, since by default, Phar Lap runs application programs at ring 0, and Eclipse runs applications at ring 3. The lower 2 bits of the selector value indicate the protection level, so these selector values are the same after screening off the lower two bits.

Table 5-3: Phar Lap and Eclipse LDT hardwired selectors.

Phar Lap Segment Selector	Eclipse Segment Selector	Description
0004H		A readable/writable data segment that points to the DOS program segment prefix (PSP) for the program.
000CH	000FH	Code selector pointing to the load image. A readable/executable code segment that points to the program. The initial selector value loaded in the CS register.

Phar Lap Segment Selector	Eclipse Segment Selector	Description
0014H	0017H	Data window on the load image. A readable/writable data segment that points to the program segment. This is the selector value initially loaded into the DS, SS, ES, FS, and GS registers (note FS exception later).
001CH	001FH	Screen. A readable/writable data segment that points to physical screen memory. This selector can be used by programs that write directly to screen memory for speed. The base address and limit of this selector are automatically updated by the DOS extender when BIOS system calls to change the video mode (Int 10H, Function 0) are made.
0024H	0027H	Program segment prefix. A readable/writable selector that is a duplicate of the descriptor that points to the program's PSP.
002CH	002FH	Pointer to environment. A readable/writable data selector that points to the DOS environment block for the program.
0034H	0037H	Base memory. A readable/writable data segment that maps the entire first megabyte of memory used by DOS.
003CH	003FH	Weitek. A readable/writable selector that maps the memory space used by the Weitek 1167 (or 3167) numeric coprocessor. If the 1167 is present, this selector is initialized and the FS register is loaded with this selector value (003CH for Phar Lap). If the Weitek coprocessor is not present, the base and limit for this selector are both set to zero, and the FS register contains the same selector value as DS.
	00B0H	Monochrome video. A readable/writable data selector that maps onto the monochrome video refresh buffer at B000:0.
	00B8H	Color video. A readable/writable data selector that maps onto the color video refresh buffer at B800:0.

The following listing provides sample code that writes directly to video memory in text mode, using the appropriate protected-mode address for each of the three environments. Graphics can be handled in a similar manner:

```
; SCREEN.ASM - Screen test program
; Based on the examples provided with Phar Lap 386|ASM/LINK
;
; This program illustrates directly accessing screen memory when
; running in 386 protected mode. Writes to screen memory in the
; text mode. Senses the DOS extender runtime dynamically and uses
; the appropriate address as follows:
;       Phar Lap        001Ch:0
;       Eclipse         001Fh:0 But 001Ch:0 will also work
;       IGC             DS:0B0000h for Monochrome, else DS:0B8000h
;
```

```
.386                                    ; generate 386 code

;       Some useful equates
CR              equ     0dh
LF              equ     0ah

IGC_ENV         equ     1
PL_ENV          equ     2
AI_ENV          equ     3

;       Screen defines
SCR_HEIGHT      equ     24              ; Screen height
SCR_WIDTH       equ     80              ; Screen width
NORMAL          equ     00700h          ; Normal attribute byte


;       Screen Memory Selectors and Offsets
PL_SCREEN       equ     01CH            ; Phar Lap selector
AI_SCREEN       equ     01Fh            ; Eclipse selector

MONO_OFFSET     equ     0B0000h         ; for IGC
COLOR_OFFSET    equ     0B8000h         ;

;       Special characters
ULCORNER        equ     0C9H            ; Double upper left corner
URCORNER        equ     0BBH            ; Double upper right corner
LLCORNER        equ     0C8H            ; Double lower left corner
LRCORNER        equ     0BCH            ; Double lower right corner
DVLINE          equ     0BAH            ; Double vertical line
DHLINE          equ     0CDH            ; Double horizontal line

dseg    segment public byte  'DATA'

scr_offset      dd      0               ; offset to screen
env             db      0               ; storage for XDOS type
_osmajor        db      0               ; storage for DOS version
_osminor        db      0
err_msg         db      'Do not know what DOS Extender we are running!'
                db      CR,LF,'$'

;       The line table
LINE_WIDTH      equ     50
LINE_CNT        equ     5

line_tab        label   byte
        db      ULCORNER,(LINE_WIDTH - 2) dup (DHLINE),URCORNER
        db      DVLINE,(LINE_WIDTH - 2) dup (' '),DVLINE
        db      DVLINE
        db      '   Screen test program for 386 protected mode   '
        db      DVLINE
        db      DVLINE,(LINE_WIDTH - 2) dup (' '),DVLINE
        db      LLCORNER,(LINE_WIDTH - 2) dup (DHLINE),LRCORNER

dseg    ends

?STACK  segment  dword stack  'STACK'
        db      8*1024 dup (?)          ; The default stack of 8K.
?STACK  ends

DGROUP  group   dseg,?STACK

cseg    segment dword public 'CODE'
CGROUP  group   cseg
        assume  cs:CGROUP,ds:DGROUP

        public  _start_,discover_env,print_string

_start_ proc    near
        call    discover_env
        or      eax,eax                 ; did we get back something
        jnz     short xdos_okay
;
        mov     ax,4C01h                ; return error code and exit
        int     21h
;
xdos_okay:
        cmp     al,IGC_ENV
        jne     short try_pharlap
;
        int     11h                     ; get equipment
        and     al,00110000b            ; screen off video
        cmp     al,30h                  ; are we on MONO screen
        mov     scr_offset,MONO_OFFSET
        xor     ax,ax
        je      short clr_screen
;
        mov     scr_offset,COLOR_OFFSET
        jmp     short clr_screen
;
try_pharlap:
        cmp     al,PL_ENV
        jne     short try_aia
;
        mov     ax,PL_SCREEN            ; Screen memory selector
        mov     es,ax                   ;    to ES.
        jmp     short clr_screen
```

```
;
try_aia:
        cmp     al,AI_ENV
        jne     short exit
;
        mov     ax,AI_SCREEN
        mov     es,ax
;
clr_screen:
        cld                             ; Set forward direction
        mov     ax,NORMAL + ' '         ; Clear the screen.
        mov     ecx,SCR_HEIGHT*SCR_WIDTH
        mov     edi,scr_offset          ; point to start of screen
        rep stosw                       ;
;
write_screen:
        mov     ax,NORMAL               ; Load normal attrib into AH,
        mov     edx,LINE_CNT            ; line count in EDX, pointer
        mov     esi,offset line_tab     ; to first text line into ESI

        mov     edi,scr_offset
        add     edi,SCR_WIDTH-LINE_WIDTH ; EDI to address screen mem
        add     edi,((SCR_HEIGHT-LINE_CNT)/2)*(SCR_WIDTH*2)

loop1:  mov     ecx,LINE_WIDTH          ; Load line width into ECX

loop2:  lodsb                           ; Move next text character to
        stosb                           ; screen with normal attrib

        inc     edi                     ; Increment screen to next char
                                        ; i.e., skip attribute byte
        dec     ecx                     ; Decrement char count and
        jne     loop2                   ; loop if not zero.

        add     edi,(SCR_WIDTH-LINE_WIDTH) * 2 ; Bump EDI to next line

        dec     edx                     ; Decrement line count and loop
        jne     loop1                   ; if not zero.

exit:   mov     ax,04C00h               ; Exit the program.
        int     21h                     ;

;;      discover_env
;       determine which DOS extender we are running under
;       uses EAX, EBX, EDX
;       env returned in EAX
;
discover_env:
        xor     eax,eax                 ; zero register used for return
        mov     ah,30h                  ; get DOS version & XDOS type
        int     21h
        mov     word ptr _osmajor,ax    ; save DOS version
;
        push    eax                     ; save EAX
        shr     eax,16                  ; IGC returns 'SG' in top
        cmp     ax,'SG'                 ; of EAX
        jne     short not_igc
;
        pop     eax
        mov     env,IGC_ENV
        jmp     short got_env
;
not_igc:
        cmp     ax,'DX'                 ; Phar Lap returns with 'DX'
        pop     eax                     ; in top 16 bits of EAX
        jne     short not_pl
;
        mov     env,PL_ENV
        jmp     short got_env
;
not_pl: push    ebx
        shr     ebx,16
        cmp     bx,'AI'
        pop     ebx
        jne     short not_any
        mov     env,AI_ENV
        jmp     short got_env
;
not_any:
        lea     edx,err_msg             ; display error and code
        call    print_string
;
got_env:
        mov     al,env                  ; return XDOS env
        movzx   eax,al                  ; zero extended to EAX
        ret
_start_ endp

print_string    proc    near            ; entry EDX=string  uses AX
        mov     ah,9
        int     21h
        ret
print_string    endp

cseg    ends

        end _start_
```

For the most part, coding in 80386 assembly language varies only slightly from 8086 practices. In the code above, all the `PUSH` and `POP` instructions use the full 32-bit registers. These are preferred over their 16-bit counterparts for two reasons: First, a slight speed penalty is paid for non-aligned memory access on the 386. Second, whereas the instruction for the full 32-bit register `PUSH` and `POP` (for example, `PUSH EAX`) is a single-byte op-code. The equivalent 16-bit `PUSH` (such as `PUSH AX`) requires an additional prefix size override byte in "USE32" code, and this increases program code size while providing no benefits.

The listing also illustrates how simple DOS calls are made from 32-bit code. Note that all pointers (for example, the pointer to the string passed to the `print_string` function) are loaded into 32-bit registers, because the value of the pointer might be greater than the 64K limit of the DX register. The `discover_env` routine determines which DOS extender the program is runinf under.

To assemble, link, and run this program under Phar Lap 386 | DOS-Extender, use the following set of DOS commands:

```
386asm screen
386link screen
run386 screen
```

As noted earlier, once you purchase a redistribution package from Phar Lap, you could bind RUN386.EXE and SCREEN.EXP together to form SCREEN.EXE that can be run directly from the DOS command line.

For Eclipse OS/386, you would use still use Phar Lap's assembler and linker, but would normally give the executable a .PLX extension. The OS/386 TSR (OS386.EXE) must already have been loaded:

```
os386
386asm screen
386link -exe screen.plx screen
up screen
```

Purchasers of a redistribution package can bind UP.EXE and SCREEN.EXP together to form an Eclipse SCREEN.EXE. This still requires that the OS386.EXE TSR be loaded separately, however. For complete transparency to the end-user, OS386.EXE can also be bound into SCREEN.EXE, though this results in a bloated file. During development, another way to run SCREEN with OS/386 is under the CP command processor:

```
os386
cp
screen
```

Finally, for IGC X-AM, you need to produce a relocatable 386 executable, using the 386LINK -RELEXE option, and a renamed copy of the IGC DOS extender (VMRUN.COM):

```
386asm screen
386link -relexe screen screen
copy vmrum.com screen.com
screen
```

Writing Into Code Segments

Another capability that is sometimes needed is the ability to write into code segments. DOS extenders provide either system calls or a selector mechanism to alias a code segment with a data segment selector.

Both the Phar Lap and the Eclipse DOS extenders use hardwired overlapping segments for the initial code and data segments. The actual selectors would be identical, except that Phar Lap applications normally run at ring 0, and OS/386 applications run at ring 3. The last 2 bits of the selector indicate the CPL (current protection level) so that OS/386 selectors will have these bits turned on. Eclipse creates an alias data selector for each code selector, using the code selector XOR 8. This same mechanism also works with the initial hardwired code and data segments set up by the Phar Lap DOS extender. The code fragment below outlines this scheme under both Phar Lap and Eclipse DOS extenders:

```
push ds                 ; save the DS register
mov ax,cs               ; get our code selector
xor ax,8                ; and convert to data selector alias
mov ds,ax
.
.
                        ; now write into our code segment
.
.
pop ds                  ; and restore our data selector
```

Installing Interrupt Handlers in Real and Protected Mode

Interrupts on the 80386/486 fall into three categories:

- hardware interrupts generated by an external hardware event
- software interrupts (commonly used for DOS and BIOS system services)
- processor exceptions generated by the 386/486 chip when memory protection or other programming errors (divide by zero, for example) are detected.

All three types of interrupts are handled in a similar manner by the DOS extenders. When an interrupt occurs in protected mode, the DOS extender always gains control unless your application has taken over an interrupt vector. Depending on the interrupt type, the DOS extender may switch the processor to the real mode (or virtual 8086 mode under IGC) and reissue the interrupt as a software interrupt. When the real-mode interrupt handler (a hardware interrupt, for example) is finished, the DOS extender switches back to protected mode and returns to the protected-mode code that was executing when the interrupt occurred. The overhead required for a 386 DOS extender to switch from protected to real mode (and from real to protected mode) can range as high as 150 microseconds. Installation of a VCPI-compatible EMS emulator can also raise the switch time (for example, one EMS emulator raised the round-trip switch time on a 16 MHz Compaq 386 from 134 ms. to 552 ms.). Faster 25 and 33 MHz machines have lower overhead.

On occasion, an application program may require control over one of the interrupt vectors. When operating in protected mode, the interrupt table is not directly accessible. In general, a program cannot get interrupt addresses by reading them from the interrupt descriptor table (IDT), nor can your application take over interrupts by writing to the interrupt table. But all of the DOS extenders support installing custom interrupt handlers. The actual function call mechanism varies from one implementation to another. Both Phar Lap and Eclipse run DOS in the real mode. Therefore, these runtimes support installing both a protected-mode and a real-mode interrupt handler. If dual handlers are installed, a shared data and variable buffer must be used from low (below 640K) memory. Writing interrupt code for both real- and protected-mode handlers accessing a shared data buffer can get a bit complicated.

Eclipse emulates DOS calls for handling protected-mode interrupt vectors using standard functions with 32-bit register conventions (set interrupt vector with AH=25H and get vector with AH=35H). As shown earlier in Table 5-2, Phar Lap uses extended function calls (set protected-mode vector with AH=2504H and get vector with AX=2502H). Both Phar Lap and Eclipse also provide extended functions to get and set interrupt handler which gain control in real mode, protected mode, or both.

The IGC runtime handles interrupts differently from the other runtimes. Because IGC runs DOS in the virtual 8086 mode, all interrupts are received by IGC protected-mode handlers (which may pass them to a virtual 8086 DOS handler). VM/RUN initially passes a global data structure (GDA) to a protected-mode application upon execution. The GDA contains pointers to two tables of interrupt vector intercepts—GDA_INTEL (the 32 lowest vectors reserved by Intel) and GDA_HINT (the remaining high interrupt vectors). GDA_HINT points to a table containing two dword entries for each interrupt vector. The first entry is a flat address pointing to a routine to be executed before interrupt processing takes place, and the second entry is a a similar flat address pointing to a routine to be executed after interrupt processing completes. A non-zero value in either slot defines an active interrupt handler (really an intercept). GDA_INTEL points to a similar table for the 32 Intel-reserved interrupts, but only the first dword entry can be active. Any intercept routines installed in these tables are actually CALLed by the IGC runtime and must use a near return (RET) instruction. No extended function calls are available to set values in these tables; an application program must explicitly add and remove table entries.

Under the Eclipse DOS extender, an application runs at the least privileged ring 3 (386/486 protection level) and the Eclipse runtime operates at the most privileged level of ring 0. Processor exception interrupts can only be vectored to a ring 0 handler. Using this scheme, the Eclipse runtime handles all processor exception interrupts and passes only software and hardware interrupts to any user-installed interrupt handler.

In contrast, the Phar Lap DOS extender runs both an application and the DOS extender at ring 0 (most privileged). Therefore, any user-installed interrupt handler must be prepared to handle processor exceptions if the Phar Lap handlers are replaced. Since hardware and processor exception interrupts overlap on the PC, this can pose additional programming difficulties. By default, the Phar Lap DOS extender relocates the hardware interrupts IRQ0-7 (Int 08-0FH) to Int 78H-7FH so hardware interrupts no longer conflict with processor exceptions. The hardware interrupts are remapped by reprogramming the Programmable Interrupt Controller (8259 PIC chip). By default, the BIOS PrintScreen handler (Int 05) is also relocated to Int 80H. This scheme improves compatibility, particularly for user-installed protected-mode handlers, since hardware interrupts can be handled separately from processor exceptions.

For most interrupt vectors of interest under Eclipse and Phar Lap runtimes, the address obtained by a GetVector function call is the address of a protected-mode surrogate for the current real-mode handler. The surrogate takes an interrupt received while the processor is in the protected mode and passes the interrupt down to a real-mode handler.

The code below illustrates functions that can be used to install an interrupt handler in protected mode under both Phar Lap and Eclipse DOS extenders:

```
; VECTOR.ASM
.386
CODE    segment dword public 'CODE'
CODE    ends

DATA    segment dword public 'DATA'

extrn   env:byte                ;flag to indicate which DOS extender
extrn   _gda:dword              ;dd storage for IGC GDA structure

DATA    ends

IGC_ENV   equu 1
PL_ENV    equ  2
AI_ENV    equ  3

public  get_vector,set_vector

cseg     segment dword public 'CODE'
CGROUP   group  cseg,CODE
        assume  cs:CGROUP,ds:DATA

;;      get_vector
;       get an interrupt vector in protected mode
;
;       Entry:  AL = vector number
;       Exit:   ES:EBX  contains old vector
;               carry flag is clear if OKAY
;
get_vector      proc
                cmp     env,AI_ENV      ;is it OS/386?
                jne     short get_pl
;
                mov     ah,35h          ;get vector address call
                int     21h             ;to ES:EBX
                clc
                ret
get_pl:
                cmp     env,PL_ENV      ;is it Phar Lap?
                jne     short get_none
;
                mov     cl,al           ;Phar Lap uses CL for vector
                mov     ax,2502h        ;Get protected-mode interrupt
                int     21h
                ret
```

```
;
get_none:       stc                             ;otherwise, it's IGC
                ret
get_vector      endp

;;      set_vector
;       set an interrupt vector in the protected mode
;
;       Entry:  DS:EDX = CS:IP for interrupt routine
;               AL     = interrupt number to set
;       Exit:   carry flag is clear if OKAY
;
set_vector      proc
                cmp     env,AI_ENV      ;is it OS/386?
                jne     short set_pl
;
                mov     ah,25h          ;set vector address call
                int     21h
                clc
                ret
set_pl:
                cmp     env,PL_ENV      ;is it Phar Lap?
                jne     short set_none
;
                mov     cl,al           ;Phar Lap uses CL for vector
                mov     ax,2504h        ;Set protected-mode interrupt
                int     21h
                ret
;
set_none:       stc                     ;otherwise, it's IGC
                ret
set_vector      endp

          cseg  ends

                end
```

Real Procedure Calls (RPCs)

Both Eclipse and Phar Lap support a mechanism to call real-mode libraries using real procedure calls. These RPC mechanisms are recommended only when source code to the real-mode procedures is not available. If source code is available, a better approach is to convert the real-mode code to run in the protected mode. This RPC mechanism is commonly used to incorporate real-mode libraries or functions that are not yet available in the protected mode (graphics and serial communications, for example). The RPC mechanism can also be used to access undocumented DOS functions and make system calls to another program, such as a network driver or SCSI device installed in memory at boot time.

In practice, a protected-mode RPC stub copies passed parameters off the stack. These variables are passed through a data (transaction) buffer with an indication of what function to call (usually a table entry number) and block moved to the RPC stub in low memory. The RPC stub in low memory recreates the stack with suitable variables, loads the CPU registers with proper values, executes the call, and passes any results back up to the protected-mode RPC stub. This feature can always be implemented by a programmer using the 386 DOS extender memory block move functions, but the DOS extenders provide this simple interface.

OS/386 has an elegant RPC mechanism that also supports sending signals from real mode up to protected mode. The Eclipse RPC scheme uses a series of very sophisticated macros to simplify the use of RPCs by programmers. Two extended functions handle real-mode procedure calls. Under OS/386, a function call is made to first initialize an RPC library (`Int 21H AH=0E0H`) and return an RPC handle. This function takes the RPC name (pointed to by DS:EDX) which may be an executable file that will be loaded by the DOS extender in the real mode. Several different RPC libraries can be initialized using this scheme, each given a unique RPC handle. Calls to real-mode libraries use this RPC handle for issuing the call to the proper procedure (`Int 21H AH=0E1H`). Macros provided by Eclipse simplify the process of passing parameters of different types, parameter counts, and invoking the proper real-mode procedure from a library. For the developer, this design dramatically simplifies problems encountered when using RPCs under OS/386. The transaction buffer for passing parameters for RPCs is limited to 4K, but this size should be ample for most real-mode procedures.

Note that inter-machine communication over a network is the model for Eclipse intermode. In fact, the term RPC is borrowed from the networking term "remote procedure call." As explained in the section in Chapter 4 on *OS/286 and the Two-Machine Model*, the two modes of the 80x86 can be viewed as two machines residing on a very local area network: that is, real mode and protected mode can communicate, but do not share address space. Just as pointers cannot be passed between machines on a network, pointers cannot be passed between the two modes on an 80x86. For this reason, any pointer parameters used in RPCs must be converted to pass by value.

The Phar Lap DOS extender has several extended functions for use in calling real-mode procedures and issuing real-mode interrupts. A mechanism is also provided to call protected-mode procedures from the real mode in an application. The intermode call buffer can vary in size from 0 to 64K bytes and is allocated from conventional memory below 640K. In comparison with the OS/386 scheme, the Phar Lap mechanism is more bare bones, and requires the developer to handle the RPC parameters more directly. Two extended function calls are supported for calling real-mode RPCs. These functions differ only in the way that segment registers are set up for the real-mode procedure. One function sets the DS register to the same value as CS (`Int 21H AX=250EH`) and allows placing values in all of the general registers before invoking the real mode procedure. The second function (`Int 21H AX=2510H`) adds support for setting arbitrary values in all of the segment registers.

Virtual Memory and Page Locking Extensions

Both Phar Lap and Eclipse support a demand-paged virtual memory option. Programs operating under a virtual memory manager (VMM) can access memory space for code and data that is larger than available physical memory in the computer. Unused sections of the applications code and data are paged to a disk swap file and automatically brought into memory when needed. If virtual memory is used, certain chunks of code may need to be locked into memory. For example, hardware interrupt routines and the DOS critical error interrupt handler must of necessity not be paged to disk. A programmer might also want to lock specific code modules used frequently within an application into memory. Both the Phar Lap and Eclipse virtual memory options support a number of extended function calls to control the virtual memory manager. Extended calls allow locking and unlocking pages in memory, freeing physical memory pages, getting memory statistics, and controlling extended and conventional memory use.

Memory allocation schemes may need to be handled differently by your application under a virtual memory manager. A program using VMM should allocate only the quantity of memory actually needed for the application. Allocating chunks of unnecessary memory only increases the size of the swap file used for paging. Ultimately, the available disk space for this swap file determines the available memory for your DOS extender application with VMM. When using a high-level language (C, Pascal, or Fortran, for example), memory allocation is generally handled by the runtime library.

Most virtual memory schemes use a variation on the least-recently used (LRU) algorithm for choosing memory to page to disk. The LRU design presumes that memory pages most recently used by an application are most likely to be referenced again soon. The least recently used chunks of memory are paged to disk when memory space is needed. Implementing a true LRU system on the 386 can be expensive in CPU time since each page reference would need to be time-stamped. The most common approximation uses a count of how frequently a page is accessed by scanning the page tables at certain time intervals. The Phar Lap virtual memory manager supports two different page replacement algorithms—least frequently used and not used recently. The Phar Lap VMM also allows installing a custom page replacement handler written by the developer.

Hardware Requirements

Three types of the 386/486 chip set are suitable for use with 386 DOS extenders. The 80386 is the original microprocessor Intel developed to follow its 80286. The 80386 has both an internal and an external 32-bit wide data bus. Memory can be read and written in 32-bit chunks.

Starting in 1989, Intel began to ship, in quantity, an 80386SX CPU that supports the 80386 instruction set, but has only a 16-bit wide external bus. This SX chip allows less expensive circuit board design, but at a reduction in performance. Memory access is limited to 16-bit chunks, and the available SX motherboards do not support use of the fast Weitek math coprocessors.

The latest entry from Intel is the 80486, which can be considered a fast 386 for most applications. A primary goal of the 80486 design was software compatibility with the 80386. The 80486 CPU has an on-chip cache unit, and the numerics coprocessor (80387) has been incorporated within the CPU. Only a few instructions were added, three of which (`INVD`, `WBINVD`, and `INVLPG`) are used for invalidating parts of the on-chip caching unit, and do not affect application software development. The other new instructions are `BSWAP` (byte swap), `XADD` (atomic exchange and add), and `CMPXCHG` (atomic compare and exchange).

In order to take advantage of the development possibilities provided by 386 DOS extenders, a PC must have an 80386 or 80486 CPU, along with some memory above the 640K that DOS normally uses. A bare minimum configuration would be a 386 machine with 2 megabytes of physical memory. A machine equipped with 4 megabytes of physical memory, however, would be a more useful development platform, especially for debugging. The memory capacity of a development machine should be easily expandable for future enhancement.

A development machine should also support both the Intel 80387 and the Weitek math coprocessor. The ideal arrangement for the 386 is separate sockets for each math chip. Daughter boards are available that plug into the extended math coprocessor 121-pin socket found on many 80386 PCs. These daughter boards provide separate sockets for both Intel and Weitek math chips. For the newer 80486 PCs with the Intel math coprocessor on-chip, a socket is needed only for the Weitek 4167 math chip. Either the 80386 with daughter board or the 80486 with a socket for the Weitek 4167 math chip will allow the development and testing of math-intensive applications for both the Intel and Weitek families of math chips.

Paging might need to be turned off on some older 386/387 machines in order to prevent a hardware lockup, due to a bug in the B stepping 16 and 20 MHz 80386 chips that occurs only when 80387 instructions are executing and paging is enabled. Many 80386 EMS emulators (such as Compaq's CEMM, Quarterdeck's QEMM, or Qualitas' 386-to-the-Max) employ paging, so this problem can show up even in what seems like real mode (actually, Virtual 8086 mode).

Another feature of older 386s is the widely-publicized multiply (MUL) bug discovered in mid-1987. Phar Lap's RUN386.EXE tests for this bug at startup, and will exit back to DOS with an error message on machines with older 386 chips. The preferred solution is to upgrade the 386 chip; a temporary work-around is to use Phar Lap's CFIG386 utility to force 386 | DOS-Extender to run anyway:

```
C:\>cfig386 run386.exe -nomul
```

Using Numeric Coprocessors with DOS Extenders

Although PC software has supplanted many applications (word processing, spreadsheets, databases, and others) from the minicomputer and mainframe world, numerics (math-intensive) applications have traditionally remained in the realm of the big machine. If a simulation takes 100 minutes of CPU time on a VAX, it can take forever on a PC. Fast, 32-bit microprocessors (Intel 80386 and Motorola 68020) alone are not sufficient to make the PC competitive with larger machines. The 32-bit microprocessors provide the ability to address large blocks of memory, but most mainframes and minicomputers have floating-point accelerators that speed math-intensive applications. Microcomputers need fast floating-point coprocessors if they are to move numerics-intensive applications off the mainframe.

The ability to produce fast numeric applications was a design goal of the 386/486 chip family. The original 80386 supported three math chips:

- 80287 coprocessor (Intel)
- 80387 coprocessor (Intel)
- 1167 chip set (Weitek).

Today, the 386/486 family also supports the following:

- 80387SX coprocessor (Intel—for 80386SX machines)
- 83D87 (Cyrix—a faster 80387)
- 80C387 (Integrated Information Technology—a faster 80387)
- 3167 coprocessor (Weitek—a 121-pin implementation of the 1167)
- 4167 coprocessor (Weitek—for the 80486).

Intel Coprocessors

Since the 80386 chip was available long before any working 387 chips, Intel decided to support the 80287 (the math chip originally designed to work with the 80286) as a stopgap measure. Of course, the 80386/80287 combination is a serious mismatch. While the 80386 chip is running at 20, 25 or 33 MHz, the 80287 is typically clocked at the AT bus speed (8 or 10 MHz), less than half the speed of the CPU.

The 80287 only provided minimal support for transcendentals, requiring software subroutines for sine, cosine, and so on. The 80287 support is more general, allowing faster in-line code to be generated by optimizing compilers.

In order to correspond with the IEEE 754 floating-point standard, of which they are a strong supporter, Intel dropped 80387 support for some minor features of the 80287.

The 80287 and 80387 generally execute the same instructions in the same number of clock cycles, but the 80387 is clocked at 16-33 MHz, versus the 7-12 MHz of the 80287.

The 80386SX machines require a special 80387SX chip that reduces numerics performance because all memory access is limited to 16-bit chunks.

Cyrix manufactures an 83D87 math chip that is a direct replacement for the Intel 80387. This chip executes a number of instructions (notably double precision and transcendentals) faster than the Intel chip. From a software standpoint, this chip looks like a faster 387. IIT manufactures an 80C387, a similar direct replacement for the Intel 80387, although it also provides some enhanced instructions.

The Intel 80486 directly incorporates the 80387 as an on-chip floating-point unit, without the need for a separate coprocessor.

Weitek Coprocessors

An original weakness of the 80386 was that the speed of the Intel math coprocessors (originally the 80287) was not competitive with the floating-point accelerators available for minicomputers and workstations. Weitek, a Silicon Valley manufacturer of high-end math, vector, and graphics processors, worked with Intel to design an 80386 interface to their workstation math chip set. Intel and Weitek jointly financed the Weitek interface development.

The Weitek WTL 1167 is a high-performance floating-point coprocessor for the 80386. The Weitek chip is more like a vector processor than an ordinary math coprocessor. Most better 80386 machines support both the 80387 and Weitek math chips by means of the Extended Math Coprocessor (EMC) socket, a 121-pin superset of the 80387 socket. The original Weitek 1167 chip set consists of three chips: the 1164 multiplier, the 1165 ALU, and the 1163 80386 interface chip, which also contains the register file. The first two chips form the core of the floating-point accelerators used in Sun, Apollo, and other workstations. The 1167 chip set is available on a small board that plugs into the 121-pin EMC socket, usually with a socket for the 80387. Weitek also manufactures the 3167 (Abacus), a single-chip, 121-pin implementation of the 1167 with some internal speed improvements. The 3167 is available in 20, 25, and 33 MHz versions. Weitek also makes a 4167 math chip for the 80486 (note that the 486 already includes an on-chip 387). In the past, Weitek and Intel have not considered the 1167 and 3167 to be in competition with the 80387; the 1167 was designed to provide faster numerics performance for high-end applications.

Commonly available 80386 machines that support the Weitek math coprocessors with the Extended Math Coprocessor socket include Acer, ALR, AT&T, Compaq, Dell, Everex, HP, Micronics, NCR, Olivetti, Tandy, and Wyse. A MicroChannel card featuring the Weitek 1167 for the IBM PS/2 Models 70 and 80 is also available from MicroWay.

To be efficient at math-intensive applications, language compilers must generate in-line code to achieve the fastest execution speed. Because of the way the Weitek math coprocessor is mapped on the 80386 (the Phar Lap and Eclipse convention versus the UNIX convention, discussed below), compilers supporting the Weitek chip must choose to generate in-line code for either one method or the other. The Weitek convention cannot be auto-sensed at runtime because inordinate amounts of patching would be required.

Handling large math-intensive applications depends on both a fast math coprocessor for floating-point numbers and fast integer and array indexing operations. Although the 80387 chip works under real-mode DOS, the Weitek math chips are fully supported only in protected mode.

The Weitek 1167 (and 3167) is memory mapped into the 80386 address space—it masquerades as a 64K block of memory in the 80386's physical memory space. On the PC, its physical address is usually `0C0000000H`; however, supporting software usually maps these addresses elsewhere. There are currently two conventions for mapping the Weitek on the PC in protected mode:

- The Phar Lap convention maps the Weitek to FS:0 (segment register FS).
- The UNIX convention maps the Weitek starting at linear address `0FFC00000H`.

If the paging capabilities of the 80386 are used, the Weitek chip can also be mapped to that notorious 64K block that begins at the address just below 1 megabyte in the virtual 8086 mode for standard DOS programs. Compaq has adopted this scheme for mapping the Weitek chip for real-mode applications, but only a handful of development tools support the Weitek chip under real-mode DOS. MetaWare High C and the Lahey DOS FORTRAN (F77L) compiler support Weitek chips under real mode.

Compaq and Weitek have defined a standard for user detection of the presence of the Weitek chip using BIOS `Int 11H`, and other manufacturers have followed Compaq's lead. The `Int 11H` BIOS routine (Equipment Check) returns EAX with bit 24 set to 1 if a Weitek chip is present. If page tables have been set up so that the device is addressable from DOS real mode, bit 23 is also set to 1. Presence of the Weitek chip can also be tested in hardware, and initializing the 1167 returns a revision number code. Would that the 80286 and 80386 could each be queried for its revision level!

All the 386 DOS extenders now support the Weitek chip. Phar Lap and Eclipse Computer Solutions employ the same convention for mapping the Weitek chip, using segmentation to FS:0. IGC implements the UNIX convention, using paging. 386 DOS extenders automatically sense the presence of a Weitek chip, using the Compaq BIOS call or hardware tests. This removes the burden from the programmer.

The Weitek 1167, 3167, and 4167 chips provide the four basic math functions, plus: negation, absolute value, comparison/testing, data movement, and format conversion functions. The Weitek 1167, 3167, and 4167 each have 32 single-precision floating-point registers (ws0-ws31) that can also be combined and used as 16 double-precision-floating point registers, along with control and status registers. Single-precision math functions take substantially less time than double-precision functions. Transcendental functions are handled by a software subroutine library provided by Weitek.

The Weitek registers are also mapped in the same 64K memory block. Therefore, context save and restore can use the fast 80386 REP MOVSD (move double word) instruction to copy or replace a block of register values. Instructions are executed by MOV instructions to this memory block. Weitek provides a set of macros for instruction mnemonics. The instructions are mapped for speed in vector operations so that multiplying an array in memory with an array in the Weitek registers can also make use of the same fast `MOVSD` instruction—except that now we point to a memory location for the Weitek multiply instruction. The lower 16 bits of the address encode the Weitek instruction type and both the source and destination registers. For example, the `WS_MUL` instruction would start at physical address `0C0000800H`, with an offset for the Weitek registers to be manipulated. The following sample instruction sequence multiplies each element of a floating point array VECTOR in 80386 memory into the corresponding elements in Weitek registers ws10 through ws29:

```
mov ecx,20                ;load the number of elements in array
mov esi,offset vector     ;point to array in memory
mov edi,offset ws_mult[t10] ;point to WFMUL address for ws10
rep movsd                 ;do the multiply into WTL registers
```

Although the compilers available to support the Weitek 1167 generate fast numeric code, hand coding crucial low-level routines using these vectorizing features can yield significant speed improvements. For math-intensive applications such as CAD, the typical speed improvement using the Weitek chip, compared with the 387, can be from 30 to 100 percent.

When considering using the Weitek math chip, keep three items in mind:

- The Weitek chip is much faster at single-precision than double-precision math.
- The divide instruction takes much longer than the multiply instruction.
- Transcendental functions are slower because they are coded as software subroutines.

If all your numerics code is double-precision transcendentals, using a Weitek chip will not provide as dramatic an improvement over the 80387 as it will with code containing lots of single-precision matrix manipulations.

Summary

In this chapter, we have showed that 32-bit protected mode is the key to tapping the power of 386 PC-compatible machines that would otherwise be used merely as "fast" 8088s. 386-based DOS extenders provide this key, without giving up MS-DOS compatibility. Many commercial programs are already using 386-based DOS extenders. As the installed base of 386 computers grows, 32-bit code will be used more extensively since most software developers must remain committed to the still-dominant XT and AT marketplace.

Even with the many features of the 386 that were discussed in this chapter, two crucial features were barely mentioned: the 386's hardware support for multitasking, and its Virtual 8086 mode. Because 386 DOS extenders exist in order to run one program at a time in 32-bit protected mode, these features of the 386 were not relevant here. In the next two chapters, we discuss two products, Microsoft Windows and Quarterdeck DESQview, that run particularly well on the 386, using its hardware support for multitasking and for multiple 8086 virtual machines.

Chapter 6

The Windows Operating Environment

Charles Petzold

The extensions to MS-DOS discussed in previous chapters were devised mainly to provide additional memory to DOS programs beyond the 640K ceiling that results from the limitations of real mode and the memory architecture of the PC.

Microsoft Windows is different. Windows is first and foremost a graphical windowing environment that runs under DOS. While Windows provides extensive memory management (including the use of protected mode in Version 3.0), you don't want to write a Windows program solely to solve your memory problems. In fact, on a system with 640K memory, a Windows program has access to less physical memory than ordinary DOS programs, due to the overhead of the Windows environment.

You should write a Windows program if you want to make use of the consistent user interface that Windows provides; if you want to draw graphics and formatted text on video displays and printers, using a device-independent graphics interface; and if you want to get a taste of what programming for the OS/2 Presentation Manager is like while targeting your program toward a wider market than OS/2 currently commands.

The memory management in Windows must be viewed as icing on the Windows cake. It is necessary because of the large memory requirements and multitasking nature of Windows, but hardly a reason to program for the environment.

In the pages ahead, we will look at the major features of Windows and examine a sample Windows program. Windows is a big system, and this discussion is hardly exhaustive. However, it should help you decide if Windows is the right alternative for your application. Additional information on Windows programming can be found in the Microsoft Windows Software Development Kit (the primary source) and several books on the subject.

Windows: A GUI for MS-DOS

Microsoft Windows is a graphical user interface (GUI) for MS-DOS. It is designed to run programs specifically written for the Windows operating environment. Programs written for Windows share the video display and other resources of the personal computer.

Multiple programs running under Windows each occupy a rectangular window on the display. The programs are characterized by a consistent user interface containing objects such as menus, buttons, and scrollbars. Windows programs can make extensive use of graphics and formatted text in a device-independent manner. Windows provides multitasking (of the non-preemptive, cooperative sort) and allows data to be exchanged among Windows programs.

Windows can also run many programs written for MS-DOS, but these programs cannot take advantage of the Windows interface or graphics. In many cases, DOS programs must run in a full-screen mode under Windows and will not be windowed or multitasked.

A History of Windows

Windows was first announced by Microsoft in November, 1983 and released two years later, in November, 1985, as version 1.01. Almost no one uses Windows 1.01 anymore, but current Windows users might find it an interesting historical curiosity. Windows 1.01 used tiled windows rather than the more common overlapping windows, and could be run in 320K of memory from two floppy disk drives.

In 1987, Microsoft extensively revised the "look and feel" of Windows, primarily to make it visually consistent with the forthcoming OS/2 Presentation Manager. In particular, overlapped windows replaced tiled windows, and an easier keyboard interface was added to menus and dialog boxes. Windows 2.0 was released in November, 1987, almost a year before the first version of the OS/2 Presentation Manager was ready.

In 1988, Windows 2.1 split into two products; the standard product became known as Windows/286. While it could still run on 8088 machines, Microsoft now recommended a 286. Windows/386 took advantage of the Virtual-86 mode of the 386 microprocessor to tame those DOS programs that wrote directly to the video display and hence could not be windowed or multitasked under previous versions of Windows.

Windows/286 and Windows/386 will merge into one Windows product when Microsoft releases Windows 3.0 in early 1990. Windows 3.0 contains a number of enhancements to earlier versions.

In particular, Windows 3.0 can take advantage of 286-compatible protected mode when running on machines using the 286 or 386 microprocessors. This gives Windows and Windows applications access to up to 16 megabytes of memory. Windows 3.0 also includes enhancements to the application program interface (API), a revamped shell that makes increased use of color and icons, as well as an attractive three-dimensional visual design.

Windows 3.0 can run in three distinct modes: real mode, standard mode, and 386 enhanced mode. Real mode requires 512K of conventional memory. Standard mode requires a 286 microprocessor and at least 256K of extended memory. Under standard mode, Windows 3.0 runs Windows applications in 286 protected mode. The 386 enhanced mode—which allows Windows to take advantage of 386 paging—requires a 386 microprocessor and at least 1 megabyte of extended memory.

Windows as a GUI

Windows is a graphical user interface, and is thus part of a tradition that began at Xerox Palo Alto Research Center (PARC) in the mid-1970s, entered the mass market with the ill-fated Apple Lisa (introduced in 1983) and the much more successful Macintosh (1984), and continues with the OS/2 Presentation Manager and UNIX-based systems such as X-Windows, Sun NeWS, OSF/Motif, and NeXT.

As the name implies, a graphical user interface provides facilities that assist programs in implementing a user interface and displaying graphics. The researchers at PARC considered the user interface to be a crucial part of a program because it is where man and machine meet. Programs that run under GUIs are often visually oriented and highly interactive.

The customary distinction between user input and program output is blurred in a GUI because graphical objects on the screen are used to obtain user input. That is, the screen itself serves as an input medium rather than simply echoing keyboard input back to the user. Manipulating objects on the screen requires the use of a pointing device such as a mouse. Windows, and many Windows applications, also have a keyboard interface that duplicates everything you can do with a mouse, but using a mouse is easier for many chores.

Because the windowing and user interface code is built into the system, programs written for a GUI can achieve a high degree of consistency in their use of common interface objects such as menus, scrollbars, and dialog boxes. This allows users to learn additional programs more easily after learning one.

Although a GUI-like windowing interface can be implemented in character mode, the support of graphics can extend the functionality of many programs considerably. For example, word processing programs can use WYSIWYG ("what you see is what you get") screen displays that mimic printer output. Database programs can allow graphics to be stored in database files along with text and numbers. Spreadsheet programs can use different fonts and display graphs.

While much of Windows is devoted to the support of the user interface and graphics, Windows also includes support for non-preemptive multitasking, memory management, RS-232 communications, and sound.

Windows and MS-DOS

Windows is often referred to as an operating *environment* because it is not in itself a full operating system. However, when Windows runs on top of MS-DOS it assumes much of the application support usually associated with operating systems.

Windows handles multitasking, memory management, user input through the keyboard and mouse, graphics output to the screen and printer, RS-232 serial communications, and sound, all without any help from MS-DOS and with very little help from the system BIOS. When Windows is running, DOS is relegated to what it does best: file I/O, other disk operations (such as changing the current directory), and a few minor chores (like maintaining the current date and time).

For these DOS services, a Windows program uses either `Int 21h` or normal C library functions that translate into `Int 21h` calls. For everything else, a Windows program uses function calls provided by Windows.

Windows can also run many programs written for MS-DOS. These are referred to in the Windows literature as *standard applications* but many Windows programmers call them *old applications* or *old apps*. Old applications are divided into two categories: *good old apps* can run in a window and be multitasked, while *bad old apps* cannot. (The word "bad" is not pejorative in this sense. Many of the best programs written for MS-DOS are bad apps when it comes to Windows.)

A good old application is a character-mode program that uses DOS and BIOS services rather than directly accessing hardware. Windows intercepts many of these DOS and BIOS calls and translates them into Windows functions. For example, when a DOS program makes BIOS video output calls to write text to the screen, Windows translates these calls into Windows functions that display a graphical rendering of the text in a window.

Bad applications are those that make use of graphics or directly access the machine's hardware. These programs must run in a full-screen mode. Windows suspends all programs currently running under Windows and removes most of itself from memory to give the DOS program as much memory space as possible.

When Windows is taking advantage of the 386 microprocessor, the distinction between good old apps and bad apps is blurred. By using the Virtual-86 mode of the 386 microprocessor, Windows can window and multitask many bad applications, even those that use graphics. However, performance of these programs is often necessarily degraded.

If you want, you can write your DOS programs so they can run in a window under any version of Windows. The basic rules are: don't use a lot of memory, and do use DOS and BIOS services rather than directly accessing hardware. However, it is rare for DOS programmers to consider windowing compatibility. It's simply not an issue.

DOS programs that themselves take advantage of protected mode are a special case. Windows 3.0 has some support for these programs, based on the mode in which Windows is running (real mode, standard mode, and 386-enhanced mode, as discussed earlier). Table 6-1 shows the ways that a DOS program can use protected mode and extended memory and still run under Windows.

Table 6-1: Allowed use of protected mode and extended memory by DOS programs.

Windows 3.0 Mode	Interface Available to DOS Application
Real	XMS
Standard	XMS and DOS Protected Mode Interface (DPMI)
386-Enhanced	DOS Protected Mode Interface

Windows does not itself have XMS support. This must be provided externally to Windows using the HIMEM.SYS driver included in the Windows retail package. The DOS Protected Mode Interface (DPMI) is provided by Windows when Windows is running in protected mode. DPMI is discussed in Chapter 9.

Programming Requirements

To program for Windows, you need a PC capable of running Windows with adequate performance. This is a 286- or 386-based personal computer with a hard disk and at least 640K of memory (preferably a megabyte or two). An EGA is adequate; a VGA is better. Strictly speaking, a mouse is not required for running Windows and many Windows applications, but you'll need a mouse to test your programs. You should also have a printer or two if you intend to write programs that print.

You will need a copy of Windows, of course, and some additional software. Although it is possible to write a Windows program in Pascal or assembly language, most programmers write them in C. C offers the greatest flexibility in handling pointers and structures, both of which show up quite a bit in Windows programming. Many C compilers do not provide the special support required for Windows programs. For this reason, you'll probably want to use Microsoft C 6.0.

You will also need the Microsoft Windows Software Development Kit (SDK). The SDK contains:

- a programmer's guide
- documentation of the Windows function calls
- the header files that declare all the Windows function calls
- import libraries necessary for linking Windows programs
- tools for creating icons, mouse pointers, dialog box templates, and fonts
- a version of the CodeView debugger suitable for debugging Windows programs.

Commitments and Trade-offs

Learning how to program for Windows requires a big commitment of time and energy. If you have no prior experience programming for a graphical user interface, the learning curve can be steep. Most programmers cite a six-month period before they become adept at Windows programming.

Moreover, there is no middle ground between programming for MS-DOS and programming for Windows. There is no such thing as a program that makes use of some Windows functions but is not an all-out Windows program.

Much of what you may have learned when programming for DOS in not applicable in Windows. You can forget about using `Int 10h` to write to the video display; you can forget about using `Int 16h` to read keystrokes; you can forget about 25 lines of 80 columns each; you can forget about using C functions such as `getch` and `printf`; you can forget about directly accessing hardware; and you can forget about intercepting interrupts.

Instead, you'll learn how to:

- make use of the Windows function calls
- structure your program to properly process messages from the Windows environment
- cooperate with the system and other Windows applications in your use of the processor, memory, and other resources
- write programs that run the same on a variety of hardware platforms
- use graphics in an attractive and meaningful manner
- design a program for the user's convenience rather than your own.

Like it or not, the graphical user interface has established itself as the standard computer interface for the 1990s. Continuing your programming career through this decade means that sooner or later you must come to grips with GUI programming and master it. You can pay your dues now or pay them later. Windows provides an excellent opportunity to pay them now.

Architecture and Features

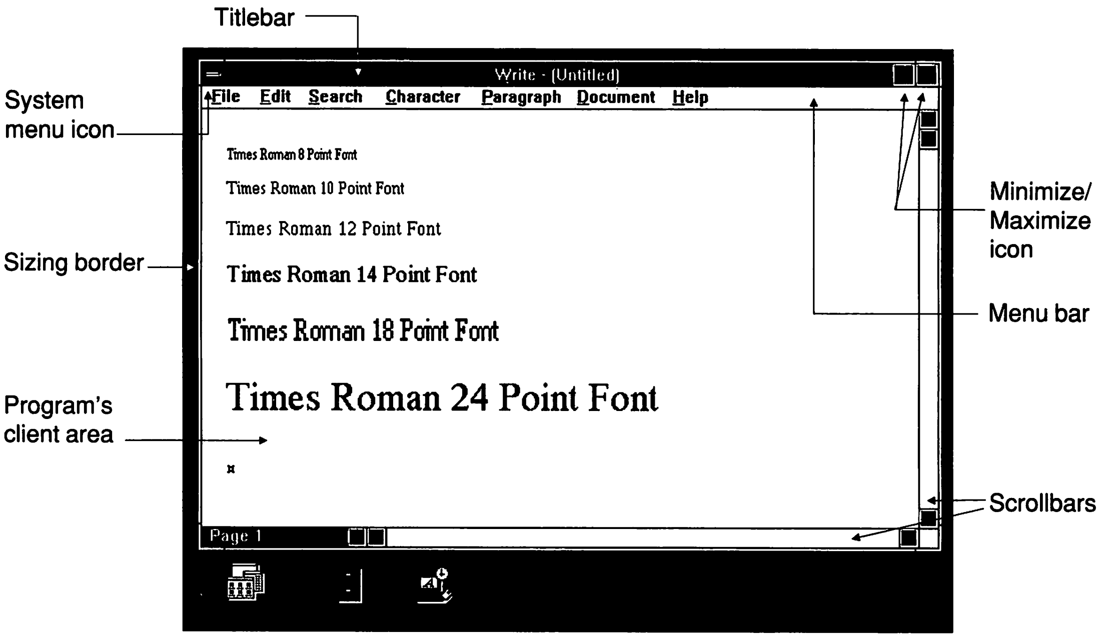
If your programming experience is limited to traditional environments such as MS-DOS or UNIX, a graphical user interface such as Windows may come as a big shock. Windows is so different from conventional environments that it influences the very structure of your programs. You will abandon a traditional top-down structure and adopt a more object-oriented structure. Indeed, Windows has often been characterized as an *object-oriented environment* with an *event-driven* or *message-driven* architecture.

The Object Called a "Window"

A Windows program creates one or more objects known as *windows*. Visually, a window is a rectangular area on the screen. The window receives user input from the keyboard and the mouse, and displays graphical program output. There are three general styles of windows: overlapped, pop-up, and child.

A Windows program generally uses an *overlapped* style for its main application window. An overlapped window usually has most or all of the window parts shown in Figure 6-1.

Figure 6-1: The parts of an overlapped window.


Titlebar
System menu icon
Sizing border
Program's client area
Minimize/ Maximize icon
Menu bar
Scrollbars
Write - (Untitled)
File Edit Search Character Paragraph Document Help
Times Roman 8 Point Font
Times Roman 10 Point Font
Times Roman 12 Point Font
Times Roman 14 Point Font
Times Roman 18 Point Font
Times Roman 24 Point Font
Page 1

A titlebar across the top of the window identifies the program. A user can move a window by grabbing the titlebar with the mouse and dragging the window to another location on the screen. To the left of the titlebar is the system menu icon. Clicking this icon with the mouse causes a menu to be displayed that lists several standard options, such as moving, sizing, or closing the window.

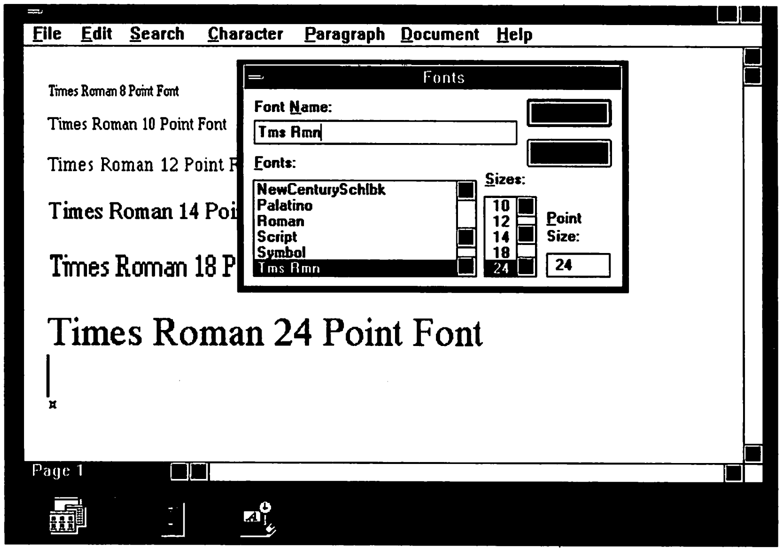
To the right of the titlebar are minimize and maximize icons. Clicking the minimize icon causes the program to be displayed as a small icon on the bottom of the screen. Clicking the maximize icon causes the window to expand to fill the entire screen.

Below the titlebar is the program's main top-level menu bar. Smaller pop-up menus are usually invoked from each item on the top-level menu. Surrounding the window is a sizing border. A user can change the size of a window by grabbing the sizing border with the mouse and dragging it. Scrollbars are often located within the sizing border on the right and bottom of the window.

Within the sizing border and scrollbars and below the program's menu is the window's *client area*. This is the area of the window in which the program displays its output. All the other areas of the window are collectively referred to as *non-client areas*.

The second style of window is the *pop-up* window. Pop-up windows are generally used for short-lived windows that a program may create, such as the dialog box shown in Figure 6-2. A pop-up window usually has a fixed size. The window may or may not have a titlebar and system menu icon. Pop-up windows do not have minimize and maximize icons.

Figure 6-2: A pop-up window used as a dialog box.



Times Roman 8 Point Font
Times Roman 10 Point Font
Times Roman 12 Point F
Times Roman 14 Poi
Times Roman 18 P
Times Roman 24 Point Font

Font Name: Tms Rmn
Point Size: 24

The third style of window is the *child* window. Child windows are generally used for small controls that take the form of buttons, text entry fields, list boxes, edit fields, and scrollbars. Most often, these appear on the surface of a dialog box, as shown in Figure 6-2. A Windows program can also place child window controls on the client area of an overlapped window.

The Window Procedure and Messages

Everything that appears on the Windows screen is a window. The user interacts with these windows. The architecture of a Windows program parallels this visual architecture.

Every window has an associated *window procedure,* which is a function that may be located either in the program that creates the window or somewhere in Windows itself. The window procedure is responsible for displaying the window on the screen and for processing keyboard and mouse input from the user.

Windows informs a window of interesting events that affect the window by sending the window *messages*. This may sound excessively abstract. What it really means is that Windows calls the window procedure, passing information about the message as parameters. Think about it for a moment: you are undoubtedly accustomed to writing programs that make calls to the operating system to perform various services. In the case of Windows, the operating environment makes calls to functions (window procedures) that are located in your program.

What are these interesting events that take the form of messages? User input certainly qualifies as an interesting event and; indeed, keyboard and mouse input messages are among the most important in Windows.

What if you grab an overlapped window's sizing border with the mouse and change the size of a window? Is that important to the window? You bet it is, and Windows has a message to tell an overlapped window when its size has been changed. Of course, when a window's size is changed, it's a good idea for the window to redraw itself, so another message indicates when a window needs repainting.

The various messages that a window procedure receives are identified by numbers, but the Windows header file defines handy identifiers that allow a program to refer to them by name. These identifiers have a prefix of WM, which stands for *window message*.

For example, `WM_KEYUP`, `WM_KEYDOWN`, and `WM_CHAR` are keyboard messages. `WM_LBUTTONDOWN` (the "L" stands for "left") and `WM_MOUSEMOVE` are mouse messages. The `WM_SIZE` message indicates a window's size has changed, `WM_PAINT` indicates that the surface of the window needs repainting, and `WM_COMMAND` indicates that the user has selected something from the program's menu. `WM_CREATE` is the first message a window procedure receives when the window is created; `WM_DESTROY` is the last message received when the window is destroyed.

When a Windows program begins execution, Windows creates a message queue for the program. This message queue is used to store messages to all the windows that the program creates. The program retrieves messages from this message queue and dispatches them to the appropriate window procedure. Most of the messages stored in the message queue are for keyboard or mouse input. These messages are said to be *posted* to the message queue. Other messages are sent directly to the window procedure from Windows.

Window procedures can also communicate among themselves, using messages. For example, dialog boxes very often contain a pushbutton or two. The pushbutton receives `WM_LBUTTONDOWN` and `WM_LBUTTONUP` messages when the user clicks the button with the mouse. The pushbutton responds by sending the window procedure for the dialog box window a `WM_COMMAND` message, indicating that the button has been pressed.

This organization of code into window procedures lends itself well to a high degree of modularity and encapsulation. A window that has a distinct appearance and performs a very specific function (such as a pushbutton) can be completely defined by a window procedure. Such window procedures are located in Windows rather than application code.

We said earlier that every window is associated with a window procedure. More precisely, every window that a program creates is based on a *window class*. The window class identifies the window procedure that processes messages to the window. This concept allows many different windows to be created, based on the same window class. For example, the pushbuttons in all Windows programs are based on the same window class and hence use the same window procedure.

The non-preemptive form of multitasking that Windows supports is also based on messages. If a Windows program attempts to retrieve a message from its message queue and the message queue is empty, Windows switches control to a program that has unprocessed messages in its queue.

The Application Program Interface

The application program interface (API) of Windows 3.0 consists of about 550 functions that Windows programs may call. These functions have descriptive names using mixed upper- and lowercase, such as `CreateWindow` and `CheckMenuItem`. All the Windows functions are declared in a large header file named WINDOWS.H, which is included in the Windows Software Development Kit. Near the top of every Windows program is the statement:

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

This includes the WINDOWS.H header file in the compilation.

Generally, a program uses these Windows functions the same way it uses C library functions in a normal C program. There are some important differences, however. First, all the Windows functions are defined with the keywords `far` and `pascal`. Second, with one oddball exception, all pointers passed as parameters to Windows functions must be *far* pointers. The function templates in WINDOWS.H defines all the Windows functions in this way; the C compiler performs any pointer conversion for you, so you usually don't have to worry about it.

The WINDOWS.H header file also defines over 70 data structures used in Windows function calls and messages, and about 1,500 defined identifiers of numeric constants. For example, the message identifiers discussed above are all defined in WINDOWS.H.

One important part of the API is the concept of the *handle*. A handle is a number that refers to an object. For example, in MS-DOS programming, a file handle is a number that refers to an open file. In Windows programming, many other objects are identified by handles. Generally, you create (or obtain access to) an object by calling a Windows function. The function returns a handle to the object. You then use the handle to refer to the object in other function calls. When you're finished using the object, you destroy (or release) it, at which time the handle becomes invalid.

For example, when a program creates a window by calling `CreateWindow`, the function returns a handle to the window. This is the most important handle in Windows. You use this handle to refer to the window when calling functions that affect the window.

Another important handle is the handle to a *device context*. The device context is the drawing surface of a window or other output device such as a printer. You need a device context to use graphics on an output device. In the sample program described later in this chapter, you'll also encounter an *instance* handle (a handle that refers to the program itself), a handle to a menu, a handle to an icon, a handle to a mouse cursor, and a handle to a brush (which is a pattern used to fill an enclosed graphical area).

Although it is not strictly part of the API, Windows programmers often use a variable naming convention that involves prefacing a variable name with a lowercase abbreviation of the data type, such as `lpsz` for a long pointer to a string terminated with a zero byte. Many of the structures defined in WINDOWS.H have field names that use this variable naming convention.

The software interface between a Windows program and the Windows operating environment is unusual for MS-DOS: first, unlike some other DOS windowing libraries, a Windows executable does not contain any code for implementing the Windows function calls. All these functions are in Windows itself. Second, unlike the case of MS-DOS and the ROM BIOS, a Windows program does not call a Windows function through a software interrupt provided in a binding library.

Instead, when you compile and link a Windows program, the calls to the Windows functions remain unresolved far calls. Windows resolves these calls to the Windows functions when the program is loaded into memory to run. This process is known as *dynamic linking*.

Dynamic Linking

Dynamic linking is an important architectural component of Windows. It is the process of resolving a function call from a program to the actual function located in a *dynamic link library* (DLL). Windows itself is mostly composed of several dynamic link libraries.

A dynamic link library is a file that contains functions that programs or other dynamic libraries may use. Like program files, a Windows dynamic link library file has a filename extension of .EXE.

Each function in a dynamic link library that can be called from outside the module is said to be *exported*. The dynamic link library's .EXE file contains a table that lists all the exported functions. Functions can be exported either by name (that is, the name of the function) or by *ordinal*, a positive number that uniquely identifies the function within the module.

When a program contains a call to a function in a dynamic link library, that function is said to be *imported* to the program. The program's .EXE file contains a table of all imported functions. The functions are identified by a module name (which is the filename of the dynamic link library without the .EXE extension) and either the function name or its ordinal. Dynamic link libraries also often make use of imported functions.

When you run a Windows program, Windows examines the list of imported functions in the program's .EXE file. It then locates the dynamic link libraries that have exported these functions, and resolves the far calls by linking the program code with the DLL code.

One of the big advantages of dynamic linking is that it allows Windows to make more efficient use of memory. If two Windows programs require the same function in the same dynamic link library, the DLL code can be shared between the two programs. It is not necessary for all the DLL code that a program requires to be loaded into memory at once. Parts of the dynamic link library can remain on disk until needed.

When you run LINK to create an executable Windows program, you make use of an *import library* included in the Windows Software Development Kit. For each Windows function a program can call, the import library identifies the module containing that function and its name or, more commonly, its ordinal number. (The ordinal numbers are preferred because they require less space in the .EXE file.) LINK uses this information to create the imported functions table in the program's .EXE file.

The three major dynamic link libraries included in Windows are: KERNEL, USER, and GDI. KERNEL contains the tasking and memory management functions, USER contains the windowing and user interface functions, and GDI contains the Windows Graphics Device Interface functions.

You can create your own Windows dynamic link libraries. This is a convenient way to share code that may be required by several different programs. Dynamic link libraries can also be products in themselves, to provide extensions to the Windows interface.

Dynamic linking is one of several architectural features developed for Windows that later found their way into OS/2.

The New EXE format

We have mentioned tables in the .EXE file. If you're familiar with the format of the MS-DOS .EXE file, you may be wondering where these tables are located.

Although Windows executables and dynamic link libraries retain the filename extension of .EXE, the files are actually a different format, called the "New Executable" format. The New Executable format is an extension of the MS-DOS .EXE format, because the New Executable file begins with the MS-DOS .EXE header and (optionally) a non-Windows MS-DOS program. Commonly, a Windows .EXE file contains an MS-DOS program that simply displays the message *This program requires Microsoft Windows* and then terminates. This is why you see this message when you attempt to execute a Windows program on the DOS command line.

The New Executable format also has a second header section, which contains an extensive amount of information that Windows uses for dynamic linking and memory management. For example, an MS-DOS .EXE file simply contains a binary image of an entire program. In the New Executable format, each code and data segment in the program is separate, and identified in a table in the second header section.

Like dynamic linking, the New Executable format is also used in OS/2, although in a slightly different format.

Real-Mode Memory Management

Memory management has always been one of the strong points of Windows, even prior to the use of protected mode in Windows 3.0. Memory can be tight in a graphical user interface, particularly when multitasking is also supported. As programs are started up and terminated, memory can become fragmented. It is therefore necessary for Windows to move blocks of memory in order to consolidate free space.

We will first examine how Windows handles memory in real mode, which is still an option when running Windows 3.0. Even under real mode, Windows implements many features that are more common to protected mode. Specifically, Windows can do the following:

- share program (and dynamic link library) code and read-only data between multiple copies of the same program (multiple copies of the same program are called *instances*)
- move code and data segments in memory to consolidate free memory space
- discard code and read-only data segments from memory (based on a least-recently-used algorithm), and later reload them from the .EXE file when necessary.

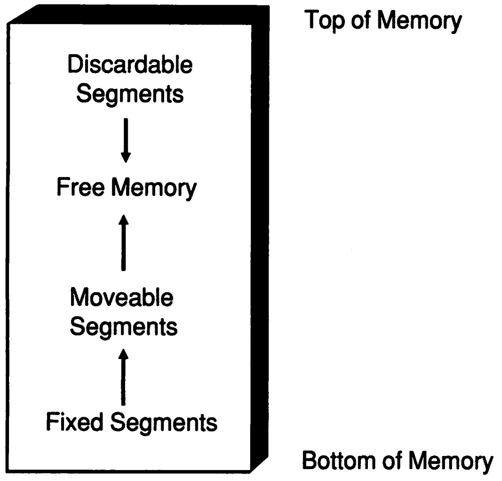
While these features are expected in protected mode, they are not easy to implement in real mode. It requires a lot of tricky code in Windows, cooperation from the programmer, some special C compiler switches, more cooperation from the programmer, the new .EXE file format, and still more cooperation from the programmer.

All memory management in Windows is based on *segments,* which (as in 16-bit protected mode) are blocks of memory that can range in size from 1 byte to 64K. A segment can be classified as *fixed* (cannot be moved in memory), *moveable* (movement is allowed to consolidate free memory space), or *discardable* (can be discarded from memory if necessary). Discardable segments are always also moveable.

The *global heap* (which is all the memory that Windows commands) is organized as shown in Figure 6-3.

Fixed segments are allocated from the bottom up. Discardable segments are allocated from the top down. Moveable segments are allocated above the fixed segments. If a new fixed segment must be allocated, and no space is available at the bottom of memory, then Windows will move the moveable segments up in memory to consolidate free space.

Figure 6-3: The organization of global memory in Windows.



It is highly recommended that Windows programs be compiled for either small model (one code and one data segment) or medium model (multiple code segments and one data segment). Because the program has only one data segment in both these models, only near (16-bit offset) pointers are required by the program to access data in that segment. These near pointers can be stored in the data segment, and they remain valid if the data segment is moved. This allows the program's data segment to be moveable.

If a program is compiled for compact or large model, the data segments must be fixed in memory. This is not recommended because it doesn't fully cooperate with Windows' memory management.

You must be careful not to store any far (32-bit segment and offset) pointers to other moveable segments, because the pointers can become invalid if the other segment is moved.

Of course, if programs were allowed only one data segment, they could not use more than 64K of data. For this reason, Windows supports the allocation of additional data segments outside the program's own data segment. That these allocated data segments can also be moveable does not, at first, seem possible under real mode. For example, if a program allocates a block of memory outside its data segment, it must store a far pointer to that segment. If Windows then moves the allocated segment in memory, the pointer is no longer valid.

To avoid this problem, Windows defines its own memory management functions. The memory allocation function does not return a pointer. Instead, it returns a handle, which is simply a number that uniquely refers to the allocated segment. The GlobalAlloc function shown below allocates a block of moveable memory 1K in length. The handle is stored in the variable hMemory:

```
hMemory = GlobalAlloc (GMEM_MOVEABLE, 1024) ;
```

Windows maintains a table in a fixed area of memory that contains these handles and the segment addresses of the memory blocks. When the segment is moved, Windows changes the entry in this table.

If the program wishes to access the allocated segment, it must call GlobalLock to lock the segment in memory. The function call returns a long pointer to the segment:

```
lpMemory = GlobalLock (hMemory) ;
```

When a program calls GlobalLock, the memory block is temporarily fixed in memory, and Windows cannot move it. When the program is finished accessing the memory, it calls the GlobalUnlock function:

```
GlobalUnlock (hMemory) ;
```

Following this function call, Windows is free to move the block of memory again. The next time the program calls GlobalLock, the pointer returned may well be different.

When the program is entirely finished using the segment, and no longer needs it, the segment can be freed:

```
GlobalFree (hMemory) ;
```

It is recommended that a Windows program call GlobalLock only when accessing a block of memory, and GlobalUnlock when it is finished accessing the memory. This should be done in the course of processing a single message. For example, a word processing program might allocate a block of memory to store the document. It would only need to lock the block when changes are being made to the document (usually on receipt of WM_CHAR messages from the keyboard or WM_COMMAND messages from the menu) or when accessing the document for other reasons (such as updating the screen during a WM_PAINT message or when printing).

Several other functions (most notably GlobalReAlloc and GlobalSize) are available to change and obtain the size of an allocated segment.

Windows does not require you to allocate only moveable segments using GlobalAlloc. You can use the GMEM_FIXED flag to allocate a segment that is always fixed in memory. In this case, GlobalAlloc returns the segment address of the allocated segment.

A GMEM_DISCARDABLE flag also exists for allocating a discardable segment. Windows is free to discard this segment from memory when it is unlocked. A segment that has been discarded by Windows is indicated by a NULL return value from GlobalLock. Usually a program does not use discardable segments unless the data in the segment can be regenerated easily. However, it is possible for a program to implement its own swapping with discardable segments. If you pass the address of a function in your program to the GlobalNotify function, Windows calls the function whenever it is about to discard a discardable segment. You can then save the contents of the memory block to disk.

Windows also includes an analogous collection of memory allocation functions named LocalAlloc, LocalLock, LocalUnlock, LocalReAlloc, LocalSize, LocalFree, and LocalNotify. These functions do for the local heap (the memory that can be allocated from the program's data segment) what the other functions do for the global heap. The organization of memory in the local heap is the same as the organization of the global heap, except that we're speaking now of memory *blocks*, and not segments. If necessary, Windows can expand the program's data segment up to 64K to accommodate a larger local heap.

Making use of these local memory allocation functions is much less important than using the global memory allocation functions, but they exist if you want to make efficient use of local memory. You can still use C memory allocation functions like malloc if you wish. The special Windows libraries cause malloc to behave like LocalAlloc called with the LMEM_FIXED parameter.

So far, we've seen how Windows is able to move data segments in memory. If you use small or medium model, do not save any long pointers in static memory, and use the global memory allocation functions, as recommended, then you are cooperating with Windows' memory management.

Code segments in Windows are generally moveable and discardable. If a code segment is discarded, Windows can reload it from the .EXE file when needed. (Windows always maintains an area in memory equal in size to the largest code segment of all the programs running under Windows. Thus, reloading the code segment never fails for space reasons.) Obviously, storing variables in a discardable code segment or writing self-modifying code leads to problems and is not recommended.

A medium model program contains multiple code segments. Functions in one code segment must make far calls to functions in the other code segments. All intersegment calls in a program are listed in a table in the .EXE file. Intersegment calls between two moveable code segments are handled by a small piece of intermediate code that Windows creates. This piece of code is called a *thunk*.

Thunks are located in a fixed area of memory. Any intersegment calls from one code segment are resolved to call a thunk. The thunk then branches to the actual function in the target code segment. When Windows moves the target code segment in memory, it adjusts the thunks accordingly. The thunk code also sets a flag whenever the thunk is called. This indicates that a particular code segment has been used. Windows uses these flags for implementing its least-recently-used algorithm to determine what code segments are candidates for discarding when memory gets low.

These thunks also take care of code segments that may be discarded from memory, or that haven't yet been loaded. When Windows discards a code segment from memory, it alters the thunk to call a function that reloads the segment from the disk file into memory. Windows then restores the thunk to its normal state and branches through the thunk again to jump to the function.

Calls from a program to a dynamic link library (DLL) are also handled through thunks. In addition, DLL entry points contain code to save the program's data segment (DS) address, load DS with the segment address of the DLL's own data segment, and restore the program's DS on exit from the function. When Windows moves a DLL data segment in memory, it must adjust the function prologues of its exported functions accordingly.

A Windows dynamic link library can call a function in a program. This occurs whenever Windows sends a message to a program's window procedure. The window procedure cannot directly load the program's DS, because multiple instances of the program may be running, and they will have different data segments. This little problem is handled by another type of thunk, which is unique to each instance of the program. The thunk loads DS with the instance's data segment address and then branches to the window procedure.

It is possible that a program may call a function in a Windows DLL, which then calls another function in another Windows DLL, which then allocates some memory that requires that the code segment in the previous DLL be moved or discarded. In this case, when the second DLL returns control to the first DLL, the code segment is gone. To compensate for this, Windows performs a trick called "walking the stack." When moving or discarding a code segment, it checks the stack to see if the code segment has been called. Windows can then adjust return addresses to the new location of the code segment, or to code that reloads the code segment in memory when it's required.

All of this may sound to you like either an extraordinary feat of software engineering or a horrible kludge. It is both. Windows memory management is ugly, confusing, and the primary source of bugs in Windows applications. Yet, it works, and nothing else in the MS-DOS world comes close to Windows in the sophistication of its real-mode memory management.

Expanded and Extended Memory Support

Beginning with version 2.0, Windows began directly supporting the Lotus/Intel/Microsoft Expanded Memory Specification version 4.0 (LIM EMS 4.0), which is discussed in detail in Chapter 2.

EMS 4.0 establishes a "bank line" in memory, above which memory is paged. With a full hardware implementation of EMS 4.0, this bank line is at the 256K mark in MS-DOS memory, which usually falls somewhere in the global heap of Windows.

EMS support is mostly transparent to Windows applications. Each Windows application (including its code segments, data segment, and additional memory allocated using the `GlobalAlloc` function) is allocated from EMS 4.0 pages, if possible. When Windows switches from one Windows application to another, it makes EMS 4.0 calls to page *out* the first application's pages, and page *in* the second application's pages.

Windows locates dynamic link library data and resources below the bank line. This allows the DLL data to be shared among all Windows applications. Additional shared data segments that a dynamic link library may need must be allocated with a special `GlobalAlloc` flag: `GMEM_NOT_BANKED`. Thunks and other data structures that must be present in a fixed area of memory at all times are also located below the bank line. Dynamic link library code, however, can be located above the bank line. In this case, multiple copies of the code can exist in several applications' EMS pages.

Windows programs can transfer data among themselves using the clipboard or dynamic data exchange (DDE). In these cases, Windows copies the data from one application's EMS pages to another. Programs that share memory in other ways, however, should use the `GMEM_NOT_BANKED` flag to allocate memory below the bank line.

In many ways, the support of protected mode in Windows 3.0 simplifies memory management. The various techniques used in previous versions of Windows to emulate protected-mode features are not required. The code to implement these techniques must still be present in Windows, however, because Windows 3.0 can be run in real mode.

Contrary to popular belief, Windows does not normally swap data segments to disk. The disk activity that occurs under Windows is due to the loading and reloading of code segments and read-only resources (such as fonts). When running in the 386 enhanced mode, however, Windows 3.0 uses the 386 paging mechanism to swap data areas.

Keyboard and Mouse Input

All user input in Windows comes through the keyboard and the mouse. A Windows program obtains this input through messages that are posted to the application's message queue. Keyboard and mouse messages are always directed to a particular window, and only one window procedure receives each message.

The window that receives keyboard input messages is the *focus window,* also called the window with *input focus.* Only one window can have the focus at any time. The user can shift focus from one application window to another, using the mouse or the Alt-Escape or Alt-Tab keys. In a dialog box, the user can shift the focus among control windows using the mouse, the Tab key, or arrow keys.

A window is responsible for indicating to the user that it has the input focus. For example, a word processing program will indicate that it has the input focus by displaying a small caret. (In other environments, the caret would be called a cursor, but that word is reserved in Windows for the bitmapped image representing the mouse.) Child window controls in a dialog box indicate they have the input focus in various ways. For example, a pushbutton displays a dotted outline around the text when it has the input focus.

Windows defines several keyboard messages, the most important of which are `WM_KEYDOWN`, `WM_KEYUP`, and `WM_CHAR`. The `WM_KEYDOWN` and `WM_KEYUP` messages occur whenever a key is pressed and released. The key is identified by a *virtual key code.* A virtual key code is defined in WINDOWS.H for every key on the keyboard. The `WM_CHAR` message is generated whenever a key (first indicated by a `WM_KEYDOWN` message) and the state of the shift keys generate an ASCII character code. The message identifies the ASCII code.

A window procedure receives mouse messages whenever the mouse cursor is positioned over the window. Pressing and releasing the left mouse button generates WM_LBUTTONDOWN and WM_LBUTTONUP messages. Similarly, the right mouse button generates WM_RBUTTONDOWN and WM_RBUTTONUP messages. Moving the mouse results in a WM_MOUSEMOVE message.

A Windows program can "capture the mouse" to continue receiving mouse input even when the mouse cursor leaves the window. For example, if you press a mouse button with the cursor positioned over a pushbutton, the pushbutton changes appearance to indicate that it has been pressed. If you keep the mouse button down and move the mouse cursor outside the pushbutton window, the window returns to normal. The pushbutton knows when the mouse cursor has left its window, because it captured the mouse.

Child Window Controls

We mentioned earlier that the architecture of a Windows program often parallels the user's visual perception of the screen. For example, a dialog box usually contains a number of input devices such as text entry fields, list boxes, scrollbars, and buttons of various sorts. The dialog box is a window, and each of these input devices is also a window. These are called *child window controls,* or *control windows,* or simply *controls.* Although controls most often appear in dialog boxes, they may also appear on the client area of a window.

Windows includes several predefined controls. These controls are predefined because Windows registers window classes for them. The window procedures are located in a Windows dynamic link library. Placing one of these controls on the client area of your window involves calling CreateWindow to create a window based on one of these preregistered classes. (It's even easier when you use controls on a dialog box. You need only define the size and placement of the controls in a dialog box template.) The window procedure then receives messages from the controls when the user interacts with them. The window procedure can also send messages to the controls.

The Static window class is very simple because windows based on this class ignore user input and send no messages. This window class displays windows that consist of text strings, frames, and filled rectangles.

The Button window class supports a variety of buttons, including radio buttons, check boxes, and pushbuttons. Each of these buttons has a distinctive appearance and contains some text.

Radio buttons are used to indicate one of several possible options, much like the buttons on car radios. The button displays a small circle and some text. Several radio buttons are grouped together. Clicking one radio button with the mouse selects that button (indicated by a filled-in circle) and deselects all the others.

Check boxes indicate program options. They consist of a square and some text. Clicking the check box with the mouse puts an X in the square. Clicking again removes it.

A pushbutton is a rectangle with text inside. Clicking the button usually indicates that the program should respond in some way. Pushbuttons are often used to dismiss dialog boxes.

You can also "press" a button using the Space Bar on the keyboard if the button currently has the input focus. Whenever a button is pressed, it sends a message to its parent window (usually the window that created the control) indicating this.

The Scrollbar window class supports vertical and horizontal scrollbars often used by applications (such as word processing programs) that display a part of a larger document in the window. By clicking various parts of the scrollbar, you can move the document within the window.

Windows based on the Edit window class are editable text-entry fields, both in singleline or multiline formats. A multiline edit control actually has much of the functionality of a rudimentary editor. The Windows NOTEPAD program is little more than a multiline edit control occupying the program's client area.

The Listbox window class supports a scrollable list of text strings. The user can select one (or, in some cases, more than one) text string using the mouse. Listboxes are commonly used for selecting a file to load into memory.

The Combobox window class is one of the enhancements to Windows 3.0. A combobox is a combination of an edit control and a listbox. The listbox is normally hidden until the user presses a little button to the right of the edit field.

In addition, you can create your own controls. If the window procedure for a custom control is located in a dynamic link library, it can be shared among multiple applications.

Graphics Device Interface (GDI)

The API of any graphical environment must include a graphics programming language. In Windows, this is known as the Graphics Device Interface, or GDI. GDI is a device-independent graphics programming language. What this means is that you use the same function calls for any graphics output device (including video displays and printers) for which a Windows device driver is present. You do not need to know the particulars of the output device; the device driver does all the translation for you.

Conceptually, a program draws graphics on *device context*, which is a drawing surface associated with a graphics output device. For drawing on the video display, a window procedure can obtain a device context for the window by calling `BeginPaint` (during the `WM_PAINT` message) or `GetDC` (during other messages). These functions return a handle to the device, which is passed as a first parameter to the GDI functions.

For printer or plotter graphics, a program obtains a device context by calling `CreateDC`. This same function can be used for creating device contexts not directly associated with an actual output device. These are the *memory* device context (useful for working with bitmaps) and the *metafile* device context (for creating a metafile).

The device context stores *attributes* that determine how the GDI drawing functions operate on the device. For example, one attribute is the color and style of the "pen" used to draw lines.

By default, all coordinates passed to GDI drawing functions are in units of pixels relative to the upper left-hand corner of the output device. However, a program can set an alternative *mapping mode* to draw in units of inches, millimeters, or arbitrary coordinates. The program can also set the coordinate origin anywhere relative to the surface of the output device.

GDI supports five basic graphics primitives:

- lines
- filled areas
- text
- bitmaps
- regions

Use the `MoveTo` and `LineTo` functions to draw a straight line. The `MoveTo` function sets the beginning of the line, and `LineTo` sets the end of the line. The `PolyLine` function draws a series of connected straight lines, and the `Arc` function draws a curved line defined by the circumference of an ellipse. All lines are drawn using an object called a *pen*. The pen defines the color of the line, its width, and its style (whether it is solid, or composed of various dashes or dots).

Windows also has several functions for filling enclosed areas. The `Rectangle` function draws a rectangle, `RoundRect` draws a rectangle with rounded corners, `Ellipse` draws an ellipse, `Chord` and `Pie` draw sections of an ellipse, and `Poly-`

`gon` fills the area enclosed by a polyline. GDI fills the area using an object called a *brush.* A brush is defined by a color and a style (such as solid or consisting of several variations of hatch marks).

The standard text output function is called `TextOut`. The text begins at a specified location on the output device. By default, a program that draws text uses the Windows *system font,* which is a variable-pitch font in Windows 3.0 and a fixed-pitch font in earlier versions. A program can select a different font for the output device (depending on what the output device supports) and obtain *text metrics* for that font. These metrics provide information about the dimensions of the font.

Bitmaps are rectangular arrays of pixels used for storing complex images. The `BitBlt` and `StretchBlt` functions can copy a bitmap from one device context to another. The `StretchBlt` function can stretch or compress a bitmap to change its size. These functions actually perform much more than simple copies: they can render a source (the bitmap) on a destination surface combined with a brush in any of 256 possible bit-wise combinations.

A *region* is a combination of rectangles, polygons, and ellipses. Regions can be filled, outlined, inverted, or used for clipping.

Graphics attribute and drawing functions may be saved in *metafiles.* These are binary coded representations of GDI functions. Metafiles can be stored in memory or on disk.

Resources

A .EXE file (either a program or a dynamic link library) almost always contains code and data segments, but it may also contain another type of segment known as a *resource segment.* Resources are read-only data that are stored in the .EXE file and loaded into memory when required. Resources are shared among multiple instances of a program.

For example, a program's icon (which is actually a small bitmap) is stored as a resource. So are any customized mouse cursors you may create. Resources also include menu templates (used to define a program's menu), dialog box templates (used to define the layout of a program's dialog boxes), keyboard accelerators (which translate certain keystrokes into menu commands), and fonts.

Interprocess Communication

To allow users to transfer data from one program to another, Windows supports a form of shared memory known as the *clipboard.*

Generally, Windows programs that work with documents have an item on the top-level menu called Edit. The Edit submenu usually has several options, including Cut, Copy, and Paste. The Cut option removes a selected area of a document and copies it to the clipboard. The Copy option copies the selected area to the clipboard without removing it from the document. Paste copies data from the clipboard to the document.

Windows defines several clipboard data formats. The three most common are ASCII text, bitmaps, and metafiles. In addition, programs can define their own clipboard formats for storing data during cut-and-paste operations. This allows a user to copy a selection from one instance of a particular word processing program to another instance without losing formatting.

An increasingly popular form of inter-process communication under Windows is Dynamic Data Exchange or DDE. DDE is a protocol rather than a specific feature of Windows. It is based on the Windows messaging system and hence requires very little in the way of additional support from Windows.

Two programs are involved in a DDE transaction. A program called the *client* wants data that may be available from another program. The program that has the data is known as the *server*. A DDE transaction begins when the client broadcasts a message to all windows running under Windows asking if they can supply data identified by some keywords. A program responding affirmatively with a message to the client becomes a server. The server can either give the client the data and end the transaction, or keep the client informed when the data changes.

Any software manufacturer who writes a program that can perform DDE server functions would tell buyers of that program the keywords required to access its data. The user can then use these keywords in any Windows application that supports DDE, perhaps in a macro language or directly in a spreadsheet or word processing field.

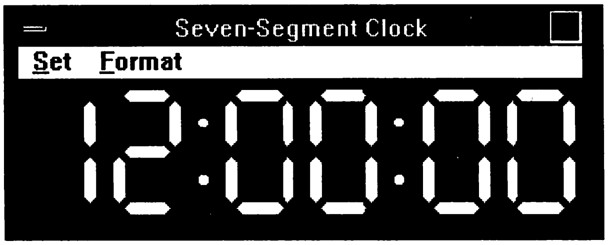
One popular demonstration of DDE involves a Windows server application that receives broadcast stock quotations via special radio hardware. A spreadsheet program can establish a DDE link and keep a spreadsheet and bar graph updated with the latest stock quotes.

A Sample Program

The best way to understand what Windows programming is all about is to examine in gory detail a complete, working Windows program.

The CLOCK7 program, shown running under Windows in Figure 6-4, shows the current time, using a simulated seven-segment LCD display. The time is updated every second. (For purposes of the non-color illustration, the program shows white numbers on a black background; the program actually displays red numbers on a black background.)

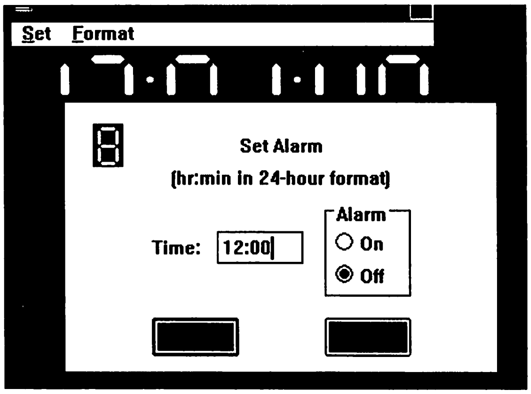
Figure 6-4: The CLOCK7 program running under Windows.



12:00:00

The menu bar has two items: Set and Format. When you select the Set option, CLOCK7 displays a drop-down menu with three options: Set Alarm, Exit, and About Clock7. The Set Alarm option displays the dialog box shown in Figure 6-5. You can enter a time, and when that time occurs, CLOCK7 displays a message box with the words "Wake up! Wake up! Wake up!" The Exit option exits the program. The About Clock7 option displays a dialog box with some information about the program.

Figure 6-5: The CLOCK7 dialog box for setting the alarm.



Time: 12:00

[✓ Off]

When you select the Format item, CLOCK7 displays a drop-down menu that lets you switch between a 12-hour format and a 24-hour format. The current selection is indicated by a check mark.

CLOCK7 illustrates many of the concepts discussed above, including window creation, message handling, resources, and graphics. If you don't understand all the workings of this program at first encounter, don't worry about it. That's normal. Windows programming is not something that can be picked up in an hour or two.

The Source Files

Programs for Windows are generally constructed from several different files. The six files required for CLOCK7 are fairly standard. You'll have similar files for almost every Windows program you write.

The CLOCK7 MAKE file is shown below:

```
#------------------
# CLOCK7 make file
#------------------

clock7.obj : clock7.c clock7.h
     cl -c -Gsw -W2 -Zp clock7.c

clock7.res : clock7.rc clock7.h clock7.ico
     rc -r clock7

clock7.exe : clock7.obj clock7.def clock7.res
     link clock7, /align:16, NUL, /nod slibcew slibw, clock7
     rc clock7.res
```

Windows programmers often use the MAKE utility (supplied with the Microsoft C Compiler) to simplify compilation and linking of the various source code files to create the executable. If you have the source code files shown here, and the Microsoft C Compiler and Windows Software Development Kit installed, you can create CLOCK7.EXE by executing:

```
MAKE CLOCK7
```

The CLOCK7 MAKE file is not only a convenient way to create CLOCK7.EXE; it also shows how the other five files contribute to the executable.

A MAKE file consists of several sections (in the case of CLOCK7, three) that begin with a left-justified line showing a target file, a colon, and one or more dependent files. If any of the dependent files has been modified more recently than the target files, MAKE runs the indented commands that follow.

The first section of CLOCK7 indicates that CLOCK7.OBJ is created from CLOCK7.C, shown below:

```
/*-----------------------------------------------------------------
   CLOCK7.C -- Seven-Segment Clock Program for Microsoft Windows
            Programmed by Charles Petzold
  ---------------------------------------------------------------*/

#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "clock7.h"
#define ID_TIMER 1

long FAR PASCAL WndProc (HWND, WORD, WORD, LONG) ;
BOOL FAR PASCAL AlarmDlgProc (HWND, WORD, WORD, LONG) ;
BOOL FAR PASCAL AboutDlgProc (HWND, WORD, WORD, LONG) ;
void DisplayTime (HDC, struct tm *, BOOL) ;
void DisplayDots (HDC) ;
void DrawDigit (HDC, int, int, int) ;

BOOL bAlarmOn = FALSE ;
int  iHour, iMin ;

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow)
     {
     static char szAppName [] = "Clock7" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS    wndclass ;

     if (!hPrevInstance)
          {
          wndclass.style          = 0 ;
          wndclass.lpfnWndProc    = WndProc ;
          wndclass.cbClsExtra     = 0 ;
          wndclass.cbWndExtra     = 0 ;
          wndclass.hInstance      = hInstance ;
          wndclass.hIcon          = LoadIcon (hInstance, szAppName) ;
          wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
          wndclass.hbrBackground  = GetStockObject (BLACK_BRUSH) ;
          wndclass.lpszMenuName   = szAppName ;
          wndclass.lpszClassName  = szAppName ;
          RegisterClass (&wndclass) ;
          }

     hwnd = CreateWindow (szAppName,         // window class name
                    "Seven-Segment Clock",   // window caption
                    WS_OVERLAPPEDWINDOW &    // window style
                         ~WS_THICKFRAME & ~WS_MAXIMIZEBOX,
                    CW_USEDEFAULT,           // initial x position
                    CW_USEDEFAULT,           // initial y position
                    CW_USEDEFAULT,           // initial x size
                    CW_USEDEFAULT,           // initial y size
                    NULL,                    // parent window handle
                    NULL,                    // window menu handle
                    hInstance,               // program instance handle
                    NULL) ;                  // creation parameters

     ShowWindow (hwnd, nCmdShow) ;
     UpdateWindow (hwnd) ;

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

long FAR PASCAL WndProc (HWND hwnd, WORD message, WORD wParam, LONG lParam)
     {
     static BOOL    b24Hour = FALSE ;
     static FARPROC lpfnAlarmDlgProc, lpfnAboutDlgProc ;
     static HANDLE  hInstance ;
     static HMENU   hMenu ;
     LPCREATESTRUCT lpcrst ;
     HDC            hdc ;
     long           lTime ;
     PAINTSTRUCT    ps ;
     POINT          pt ;
     RECT           rect ;
     struct tm      *datetime ;

     switch (message)
          {
          case WM_CREATE:
               lpcrst = (LPCREATESTRUCT) lParam ;
               hInstance = lpcrst->hInstance ;

               lpfnAlarmDlgProc = MakeProcInstance (AlarmDlgProc, hInstance) ;
          lpfnAboutDlgProc = MakeProcInstance (AboutDlgProc, hInstance) ;

          hdc = GetDC (hwnd) ;
          SetMapMode (hdc, MM_HIENGLISH) ;
          pt.x =  4000 ;
          pt.y = -1000 ;
          LPtoDP (hdc, &pt, 1) ;
          ReleaseDC (hwnd, hdc) ;

          rect.left   = 0 ;
          rect.top    = 0 ;
          rect.right  = pt.x ;
          rect.bottom = pt.y ;
          AdjustWindowRect (&rect, WS_OVERLAPPEDWINDOW & ~WS_THICKFRAME,
                            TRUE) ;

          MoveWindow (hwnd, lpcrst->x, lpcrst->y, rect.right - rect.left,
                      rect.bottom - rect.top, FALSE) ;

          hMenu = GetMenu (hwnd) ;
          SetTimer (hwnd, ID_TIMER, 1000, NULL) ;
          return 0 ;

     case WM_COMMAND:
          switch (wParam)
               {
               case IDM_HOUR12:
                    CheckMenuItem (hMenu, IDM_HOUR12, MF_CHECKED) ;
                    CheckMenuItem (hMenu, IDM_HOUR24, MF_UNCHECKED) ;
                    b24Hour = FALSE ;
                    return 0 ;

               case IDM_HOUR24:
                    CheckMenuItem (hMenu, IDM_HOUR12, MF_UNCHECKED) ;
                    CheckMenuItem (hMenu, IDM_HOUR24, MF_CHECKED) ;
                    b24Hour = TRUE ;
                    return 0 ;

               case IDM_ALARM:
                    DialogBox (hInstance, "AlarmBox", hwnd,
                               lpfnAlarmDlgProc) ;
                    return 0 ;

               case IDM_ABOUT:
                    DialogBox (hInstance, "AboutBox", hwnd,
                               lpfnAboutDlgProc) ;
                    return 0 ;
                    case IDM_EXIT:
                         SendMessage (hwnd, WM_CLOSE, 0, 0L) ;
                         return 0 ;
                    }
               break ;

          case WM_TIMER:
               time (&lTime) ;
               datetime = localtime (&lTime) ;

               if (datetime->tm_hour == iHour && datetime->tm_min == iMin
                         && bAlarmOn == TRUE)
                    {
                    bAlarmOn = FALSE ;
                    MessageBox (hwnd, "Wake Up! Wake Up! Wake Up!",
                                "Alarm", MB_OK | MB_ICONASTERISK) ;
                    }

               hdc = GetDC (hwnd) ;
               DisplayTime (hdc, datetime, b24Hour) ;
               ReleaseDC (hwnd, hdc) ;
               return 0 ;

          case WM_PAINT:
               time (&lTime) ;
               datetime = localtime (&lTime) ;

               hdc = BeginPaint (hwnd, &ps) ;
               DisplayTime (hdc, datetime, b24Hour) ;
               DisplayDots (hdc) ;
               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY:
               KillTimer (hwnd, ID_TIMER) ;
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
     }

BOOL FAR PASCAL AlarmDlgProc (HWND hwnd, WORD message, WORD wParam, LONG lParam)
     {
     static BOOL bLocalAlarmOn ;
     static char szAlarmTime[10] = "12:00" ;
     char        szParseTime[10] ;

     switch (message)
```

```
          {
          case WM_INITDIALOG:
               SendDlgItemMessage (hwnd, IDD_ALARMTIME, EM_LIMITTEXT, 6, 0L) ;
               SetDlgItemText (hwnd, IDD_ALARMTIME, szAlarmTime) ;
               CheckRadioButton (hwnd, IDD_ALARMON, IDD_ALARMOFF,
                                   bAlarmOn ? IDD_ALARMON : IDD_ALARMOFF) ;
               bLocalAlarmOn = bAlarmOn ;
               return TRUE ;

          case WM_COMMAND:
               switch (wParam)
                    {
                    case IDD_ALARMON:
                    case IDD_ALARMOFF:
                         bLocalAlarmOn = (wParam == IDD_ALARMON) ;
                         CheckRadioButton (hwnd, IDD_ALARMON, IDD_ALARMOFF,
                                             wParam) ;
                         return TRUE ;


                    case IDOK:
                         GetDlgItemText (hwnd, IDD_ALARMTIME, szAlarmTime, 10) ;
                         strcpy (szParseTime, szAlarmTime) ;
                         iHour = atoi (strtok (szParseTime, ":")) ;
                         iMin  = atoi (strtok (NULL, " :")) ;

                         if (iHour < 0 || iHour > 23 || iMin < 0 || iMin > 59)
                              {
                              MessageBox (hwnd, "Time is not valid!", NULL,
                                             MB_OK | MB_ICONEXCLAMATION) ;
                              SetFocus (GetDlgItem (hwnd, IDD_ALARMTIME)) ;
                              return TRUE ;
                              }

                         bAlarmOn = bLocalAlarmOn ;
                         EndDialog (hwnd, TRUE) ;
                         return TRUE ;

                    case IDCANCEL:
                         EndDialog (hwnd, FALSE) ;
                         return TRUE ;
                    }
               break ;
          }
     return FALSE ;
     }
BOOL FAR PASCAL AboutDlgProc (HWND hwnd, WORD message, WORD wParam, LONG lParam)
```

```
     {
     switch (message)
          {
          case WM_INITDIALOG:
               return TRUE ;

          case WM_COMMAND:
               switch (wParam) ;
                    {
                    case IDOK:
                    case IDCANCEL:
                         EndDialog (hwnd, 0) ;
                         return TRUE ;
                    }
               break ;
          }
     return FALSE ;
     }

void DisplayTime (HDC hdc, struct tm *datetime, BOOL b24Hour)
     {
     SetMapMode (hdc, MM_HIENGLISH) ;
     SetWindowOrg (hdc, 0, 1000) ;

     if (!b24Hour)
          if ((datetime->tm_hour %= 12) == 0)
               datetime->tm_hour = 12 ;

     DrawDigit (hdc, 100, 100,  datetime->tm_hour / 10 != 0 ?
                                datetime->tm_hour / 10 : 10) ;
     DrawDigit (hdc,  700, 100, datetime->tm_hour % 10) ;
     DrawDigit (hdc, 1500, 100, datetime->tm_min / 10) ;
     DrawDigit (hdc, 2100, 100, datetime->tm_min % 10) ;
     DrawDigit (hdc, 2900, 100, datetime->tm_sec / 10) ;
     DrawDigit (hdc, 3500, 100, datetime->tm_sec % 10) ;
     }

void DrawDigit (HDC hdc, int x, int y, int iDigit)
     {
     static BOOL bSegmentOn[11][7] = { 1, 0, 1, 1, 1, 1, 1,    // 0
                                       0, 0, 0, 0, 1, 0, 1,    // 1
                                       1, 1, 1, 0, 1, 1, 0,    // 2
                                       1, 1, 1, 0, 1, 0, 1,    // 3
                                       0, 1, 0, 1, 1, 0, 1,    // 4
                                       1, 1, 1, 1, 0, 0, 1,    // 5
                                       1, 1, 1, 1, 0, 1, 1,    // 6
                                       1, 0, 0, 0, 1, 0, 1,    // 7
                                       1, 1, 1, 1, 1, 1, 1,    // 8
```

```
                                        1, 1, 0, 1, 1, 0, 1,    // 9
                                        0, 0, 0, 0, 0, 0, 0 } ; // blank
     static int   iSegmentType[7]  = {  0, 0, 0, 1, 1, 1, 1 } ;
     static LOGBRUSH logbrBlack    = {  BS_SOLID, RGB(0,0,0),   0 } ;
     static LOGBRUSH logbrRed      = {  BS_SOLID, RGB(255,0,0), 0 } ;
     static POINT ptSegOrigin[7]   = {    0, 800, 0, 400,   0, 0, 0, 400,
                                        400, 400, 0,   0, 400, 0 } ;
     static POINT ptSegment[2][6]  = {   25,   0,   75, 50,  325,   50,
                                        375,   0,  325,-50,   75,  -50,
                                          0,  25,   50, 75,   50,  325,
                                          0, 375,  -50,325,  -50,   75 } ;
     HBRUSH         hbrBlack, hbrRed ;
     int            iSeg ;

     hbrBlack = CreateBrushIndirect (&logbrBlack) ;
     hbrRed   = CreateBrushIndirect (&logbrRed) ;
     SaveDC (hdc) ;
     OffsetWindowOrg (hdc, -x, -y) ;

     for (iSeg = 0 ; iSeg < 7 ; iSeg++)
          {
          SaveDC (hdc) ;
          OffsetWindowOrg (hdc, -ptSegOrigin[iSeg].x, -ptSegOrigin[iSeg].y) ;
          SelectObject (hdc, bSegmentOn[iDigit][iSeg] ? hbrRed : hbrBlack) ;
          Polygon (hdc, ptSegment[iSegmentType[iSeg]], 6) ;
          RestoreDC (hdc, -1) ;
          }

     SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
     RestoreDC (hdc, -1) ;
     DeleteObject (hbrBlack) ;
     DeleteObject (hbrRed) ;
     }

void DisplayDots (HDC hdc)
     {
     static LOGBRUSH logbrRed = { BS_SOLID, RGB(255,0,0), 0 } ;
     HBRUSH            hbrRed ;

     hbrRed = CreateBrushIndirect (&logbrRed) ;
     SelectObject (hdc, hbrRed) ;

     Ellipse (hdc, 1250, 350, 1350, 250) ;
     Ellipse (hdc, 1250, 750, 1350, 650) ;
     Ellipse (hdc, 2650, 350, 2750, 250) ;
     Ellipse (hdc, 2650, 750, 2750, 650) ;
     SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
     DeleteObject (hbrRed) ;
     }
```

CLOCK7.C requires the CLOCK7.H file shown below:

```
/*----------------------
   CLOCK7.H header file
  ----------------------*/

#define IDM_HOUR12       1
#define IDM_HOUR24       2
#define IDM_ALARM        3
#define IDM_EXIT         4
#define IDM_ABOUT        5

#define IDD_ALARMTIME    10
#define IDD_ALARMON      11
#define IDD_ALARMOFF     12
```

The compiler flags shown in CLOCK7 are normal for compiling a Windows program. In particular, the -Gsw switch (which is actually two switches, -Gs and -Gw) inhibits stack checks and causes the compiler to create a special prologue on far functions necessary for loading the program's data segment on entry to a window procedure.

The second section of the CLOCK7 make file shows that CLOCK7.RES (a compiled resource file) is created from CLOCK7.RC (the resource script shown below), CLOCK7.H, and CLOCK7.ICO by running the RC.EXE resource compiler included with the Windows Software Development Kit.

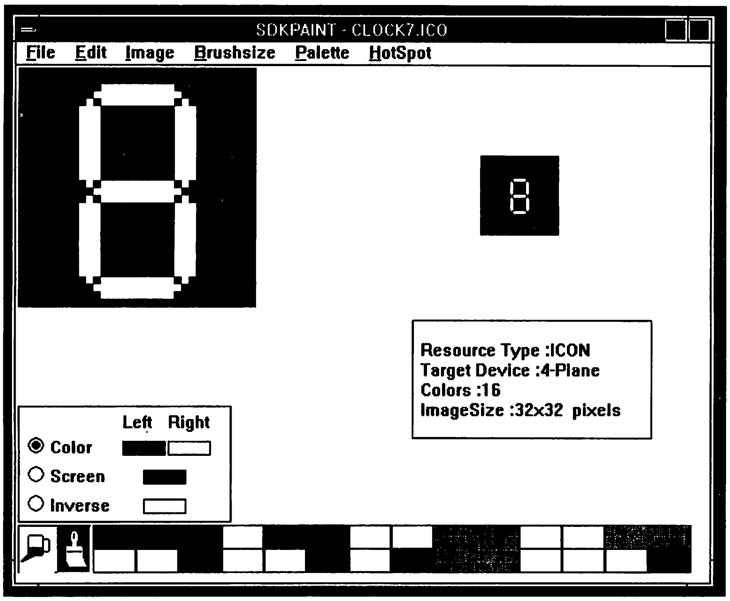
```
/*---------------------------
   CLOCK7.RC resource script
  ---------------------------*/

#include <windows.h>
#include "clock7.h"

Clock7 ICON clock7.ico

Clock7 MENU
     {
     POPUP "&Set"
          {
          MENUITEM "&Set Alarm...",     IDM_ALARM
          MENUITEM SEPARATOR
          MENUITEM "E&xit",             IDM_EXIT
          MENUITEM SEPARATOR
          MENUITEM "A&bout Clock7...",  IDM_ABOUT
          }
     POPUP "&Format"
          {
          MENUITEM "&12 Hour",          IDM_HOUR12, CHECKED
          MENUITEM "&24 Hour",          IDM_HOUR24
          }
     }

AlarmBox DIALOG 20, 20, 160, 100
   STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT "Set Alarm"                   -1,    0, 12  160,   8
     CTEXT "(hr:min in 24-hour format)" -1,    0, 24, 160,   8
     ICON  "Clock7"                      -1,    8,  8,   0,   0
     LTEXT "Time:"                       -1,    3, 50,  20,   8
     EDITTEXT               IDD_ALARMTIME,   56, 48,  32,  12,  ES_AUTOHSCROLL
     GROUPBOX "Alarm"                    -1,   96, 36,  32,  36,
     RADIOBUTTON "On"          IDD_ALARMON,  100, 46,  24,  12,  WS_GROUP
     RADIOBUTTON "Off"        IDD_ALARMOFF,  100, 58,  24,  12
     DEFPUSHBUTTON "OK"                IDOK,   32, 80,  32,  14,  WS_GROUP
     PUSHBUTTON "Cancel"           IDCANCEL,   96, 80,  32,  14,  WS_GROUP
     }

AboutBox DIALOG 20, 20, 160, 80
   STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT "Clock7"                          -1,  0, 12, 160,  8
     ICON  "Clock7"                          -1,  8,  8,   0,  0
     CTEXT "Seven-Segment Clock"             -1,  0, 36, 160,  8
     CTEXT "Programmed by Charles Petzold"   -1,  0, 48, 160,  8
     DEFPUSHBUTTON "OK"                    IDOK, 64, 60,  32, 14, WS_GROUP
     }
```

CLOCK7.ICO is a binary file containing an image of the program's icon. Figure 6-6 shows CLOCK7.ICO as it appears in the SDKPAINT program supplied with the Windows Software Development Kit. The icon is simply a "figure 8."

The third section of the CLOCK7 make file shows that CLOCK7.EXE is created from CLOCK7.OBJ, CLOCK7.RES, and CLOCK7.DEF (the module definition file shown below). This involves running the LINK.EXE linker with the special C libraries. The last parameter indicates the CLOCK7.DEF file. The RC.EXE resource compiler is then run again to add the compiled resources to CLOCK7.EXE.

```
;----------------------------------
; CLOCK7.DEF module definition file
;----------------------------------

NAME           CLOCK7

DESCRIPTION    'Seven-Segment Clock Program for Microsoft Windows'
EXETYPE        WINDOWS
STUB           'WINSTUB.EXE'
CODE           MOVEABLE
DATA           MOVEABLE MULTIPLE
HEAPSIZE       1024
STACKSIZE      4096
EXPORTS        WndProc
               AlarmDlgProc
               AboutDlgProc
```

Figure 6-6: The CLOCK7.ICO file shown in SDKPAINT.



Resource Type :ICON
Target Device :4-Plane
Colors :16
ImageSize :32x32 pixels

The CLOCK7.C File

CLOCK7.C contains all the source code for the program. It begins by including the WINDOWS.H header file:

```
    #include windows.h
```

Also included are a few normal C header files and CLOCK7.H. CLOCK7.C has three global variables defined near the top of the file: `bAlarmOn`, `iHour`, and `iMin`. These indicate the time for the alarm.

CLOCK7 calls some 40 different Windows functions, and much of this discussion will concentrate on what these function calls do.

The WinMain Function

Just as `main()` is the entry point to a conventional C program, a function called `WinMain` is the entry point to a Windows program. CLOCK7's `WinMain` function is fairly standard. Similar code appears in almost every Windows program.

`WinMain` has four parameters: the first (`hInstance`) is called an *instance handle*. This is a number that uniquely identifies the program in Windows. It is equivalent to a "task ID" or "process ID" in other operating systems. The second parameter is `hPrevInstance`, the instance handle of the most previously executed instance of the program still running under Windows. If no other copy of the program is running under Windows, `hPrevInstance` equals NULL (0).

The `lpszCmdParam` parameter to `WinMain` is not used in CLOCK7. This is a zero-terminated string that contains any parameters passed to the program on a command line. The last parameter (`nCmdShow`) is a number that indicates whether CLOCK7 is to be displayed initially as a normal window or as an icon.

The first job for `WinMain` is registering a window class for the program's window. This requires setting the fields of a structure of type `WNDCLASS` (a structure defined in WINDOWS.H) and passing a pointer to the structure to the `RegisterClass` function. Because all instances of the same program can share the same window class, this need only be done if `hPrevInstance` is not equal to NULL.

The two most important fields of the `WNDCLASS` structure are the second and the last. The second field is `lpfnWndProc`, which is set to the function `WndProc`, the window procedure to be used for all windows based on this window class. The last field is `lpszClassName`, the name of the window class. This is set to the string "Clock7" (stored in `szAppName`).

The `WNDCLASS` structure also defines the icon, cursor, background color, and menu to be used for all windows based on this window class. The `LoadIcon` function loads the icon from the resource segment of the CLOCK7.EXE file and returns a handle to the icon. CLOCK7's icon is defined by the name "Clock7" in the CLOCK7.RC resource script.

The LoadCursor function loads one of the standard Windows mouse cursors (the arrow cursor) and returns a handle to the cursor. This cursor appears when the mouse is positioned over CLOCK7's window. The GetStockObject function returns a handle to a solid black brush pattern. The lpszMenuName field is set to the name of the program's menu (in this case, "Clock7") as defined in the CLOCK7.RC resource script.

After CLOCK7 registers the window class, it can create a window by calling CreateWindow. The first parameter indicates the window class, the second parameter is the text that will appear in the window's titlebar, and the third parameter defines the style of the window, using identifiers from WINDOWS.H, beginning with the WS ("window style") prefix. The most common window style is WS_OVERLAPPEDWINDOW. This identifier is actually a combination of several other WS identifiers, each of which is a bit flag indicating one of the components of the application's window. CLOCK7 doesn't have a sizing border or maximize box, so these bit flags are eliminated by a bitwise AND operation with the inverses of WS_THICKFRAME and WS_MAXIMIZEBOX.

CreateWindow returns a handle to the window. CLOCK7 uses this handle in the next two function calls:

The ShowWindow call displays the window on the screen. The first parameter is hwnd, and the second is nCmdShow, the last parameter to WinMain. The value of nCmdShow depends on whether the program was run normally, or loaded to be displayed initially as an icon. The call to UpdateWindow instructs the window to paint itself.

WinMain then enters the standard Windows message loop. The message loop consists of calls to GetMessage, TranslateMessage, and DispatchMessage in a while loop. GetMessage retrieves the next message from the program's message queue and stores the message in a structure of type MSG. TranslateMessage performs some keyboard translation, and DispatchMessage sends the message to a window procedure for processing.

GetMessage returns a non-zero value for every message except one: a special message named WM_QUIT. WM_QUIT indicates that the program is terminating. The program drops out of the message loop and WinMain ends, ending the program. (We'll discuss how the WM_QUIT message gets into the message queue later.)

The WndProc Window Procedure

It is unusual for a C program to contain a function that is not directly called from the program. Yet this is the case with the function named WndProc. WndProc is the window procedure for CLOCK7's application window. The function is called from Windows when sending messages to that window.

A window procedure has four parameters. These parameters are the first four fields of the MSG structure used in WinMain to retrieve messages from the message queue and dispatch them to a window procedure.

The first parameter, hwnd, is the handle to the window that is to receive the message. In CLOCK7, this is the same value returned from CreateWindow. When multiple windows are based on the same window class (and hence use the same window procedure) this parameter allows the window procedure to identify the particular window receiving the message.

The second parameter to WndProc is named message. This is a number that identifies the message. All the messages have constant identifiers defined in WINDOWS.H, beginning with the prefix WM ("window message"). The last two parameters (wParam and lParam) are called *message parameters*. These parameters provide more information about particular messages. Their values, and how they are interpreted, depend on the message they accompany.

Programmers who write for Windows often use a switch and case construction to process messages sent to the window procedure. CLOCK7 processes five messages: WM_CREATE, WM_COMMAND, WM_TIMER, WM_PAINT, and WM_DESTROY (each of which is described below). Generally, after processing a message, the window procedure returns zero.

WndProc receives many more messages besides these five, but it doesn't process any of them. Any message that a window procedure chooses to ignore must be passed to DefWindowProc for default processing. This occurs near the bottom of WndProc in CLOCK7.C.

Calling DefWindowProc for all unprocessed messages is important. In particular, this function processes all messages involving non-client areas of the window. For example, the WM_NCPAINT message instructs the window procedure to paint the non-client areas of the window, such as the titlebar and the sizing border. WndProc doesn't want to bother with this job, so it passes the message on to DefWindowProc. DefWindowProc also processes keyboard and mouse input to the non-client areas. This is how a program's menu can be functional without requiring any code to handle keyboard and mouse input to the menu.

Local variables defined in the window procedure can be either static or automatic. The static variables are those that must retain their values between messages. You can use automatic variables for anything that is required only during the course of a single message.

The WM_CREATE Message

The WM_CREATE message is the first message the window procedure receives. This is usually a good time to perform window initialization.

The WM_CREATE message is sent to the window procedure when the program calls CreateWindow in WinMain to create the window: control passes from WinMain to the CreateWindow function in Windows, to WndProc in CLOCK7 to process WM_CREATE, back to Windows to finish the CreateWindow call, and then back to WinMain. By the time CreateWindow returns, WndProc has already processed the WM_CREATE message.

During the WM_CREATE message, the lParam parameter to the window procedure is a long pointer to a structure of type CREATESTRUCT. This structure contains all the data passed as parameters to the CreateWindow function. WndProc casts lParam to an LPCREATESTRUCT type (defined in WINDOWS.H as a long pointer to a structure of type CREATESTRUCT) and stores it in a variable named lpcrst. WndProc then obtains the hInstance handle from this structure. The next two lines of code in WndProc require a little background:

We discussed earlier how Windows is able to move a code segment in memory even if the segment contains a function that is called from outside the segment with a far call. There are three such functions in CLOCK7. The first is WndProc, which is called from Windows whenever Windows sends it a message. The other two are AlarmDlgProc and AboutDlgProc, which are dialog box procedures (very similar to window procedures) for the two dialog boxes.

Windows must create a thunk for the window procedures and dialog box procedures. This thunk is located in a fixed area of memory. The thunk sets the correct data segment address for the instance of the program and branches to the actual function in the moveable segment.

The calls to MakeProcInstance during the WM_CREATE message create the thunks for the dialog box procedures. The addresses of the two dialog box procedures are passed as parameters to the MakeProcInstance function. The function returns the address of the thunk. WndProc saves these addresses in static memory for later use when invoking the dialog boxes.

The next job for WM_CREATE is somewhat messy. Usually, an application window has a sizing border that the user can drag to change the size of the window. CLOCK7, however, has a window with a fixed size, with a client area four inches wide by one inch high.

When CLOCK7 creates the window in WinMain by calling CreateWindow, the CW_USEDEFAULT identifier is used to specify that Windows is responsible for initially positioning and sizing the window. This default size and position can be overridden by a call to MoveWindow.

The size of the window, however, must be specified in units of pixels, so the four inch by one inch size must be converted to pixels. The size in pixels will be different, depending on the type of video adapter, so the conversion must be done in a device-independent manner.

To do this, WndProc first obtains a device context for the window by calling GetDC. This is the normal way to obtain a device context when processing a message other than WM_PAINT. WndProc then sets the mapping mode of the device context to MM_HIENGLISH using the SetMapMode function. This causes the device context to have logical coordinates of 0.001 inches.

WndProc sets the two fields of a POINT structure named pt to the desired size of the client window in units of 0.001 inches. The x field is set to 4000 and the y field is set to -1000. (The minus sign is required because of a different vertical orientation between MM_HIENGLISH units and pixels.) This POINT structure is passed to LPtoDP, which converts logical points (in units of MM_HIENGLISH) to device points (pixels). The device context is then released by calling ReleaseDC.

Now we have the size of the client area in units of pixels, and we're almost there. What the MoveWindow function requires is the size and position of the whole window, not just the client area. This is the job of AdjustWindowRect, which can convert a client area position and size specified in a RECT (rectangle) structure to a full window position and size. The MoveWindow function then positions and sizes the window.

There are two more much smaller jobs left for the WM_CREATE message. GetMenu obtains a handle to the window's menu. This is stored in a static variable and later used when processing WM_COMMAND messages. The SetTimer function sets the Windows timer. The third parameter, of 1000, indicates that Windows should post a WM_TIMER message to the window procedure once every 1,000 milliseconds, or one second.

The WM_COMMAND Message

A window procedure receives a WM_COMMAND message when the user selects an option from the program's menu. What really happens is that DefWindowProc handles all keyboard and mouse input to the menu, and then sends the window a WM_COMMAND message when the user selects an option.

The menu is defined as a template in CLOCK7.RC, the resource script file. The template begins with the name of the menu (in this case "Clock7") and the MENU keyword. This name was assigned to the lpszMenuName field of the WNDCLASS structure prior to calling RegisterClass in main(). That's how the menu becomes part of the CLOCK7 window.

The menu template is fairly self-explanatory. The POPUP keyword is for an item on the main menu bar that invokes a pop-up menu. The pop-up menu is defined by a series of MENUITEM statements that list the options on the pop-up menu. An ampersand (&) in the text string of each item specifies which letter is to be underlined and used for the menu's keyboard interface.

All the items a user can select for a program command are associated with an identifier beginning with the prefix IDM, which stands for "ID for a Menu item." These identifiers are defined in CLOCK7.H. When the user selects one of these items, the window procedure receives a WM_COMMAND message with the wParam parameter set to the value of the identifier. Generally, a window procedure uses another switch and case construction to process menu selections, based on the value of wParam.

The first two menu items processed in the WM_COMMAND section of WndProc are IDM_HOUR12 and IDM_HOUR24, which indicate the user has selected the "12 Hour" or "24 Hour" time format, respectively. The program must place a check mark next to the selected option and remove the check mark from the other option. WndProc does this by calls to CheckMenuItem. It then sets the static variable b24Hour to FALSE or TRUE, depending on which option the user selected.

A wParam value of IDM_ALARM indicates that the user has selected the "Set Alarm" option. The program must respond by displaying a dialog box. This is the job of the DialogBox function. The "AlarmBox" parameter refers to the name of the dialog box template defined in the CLOCK7.RC resource script. The lpfn-AlarmDlgProc is the address returned from the MakeProcInstance call in WM_CREATE. This is the address of the thunk for the AlarmDlgProc function in CLOCK7.C. The DialogBox function does not return until the dialog box has been dismissed.

Similarly, a wParam value of IDM_ABOUT means that the user wants to see the program's "About" dialog box. The call to DialogBox specifies the "AboutBox" template and the thunk for the AboutDlgProc dialog procedure.

Finally, a wParam value of IDM_EXIT indicates that the user has selected the "Exit" option. WndProc does something very strange in response to this message: it calls SendMessage to send itself a WM_CLOSE message. This means that WndProc is called recursively. But it doesn't process the WM_CLOSE message! DefWindowProc, however, does, and it responds to the WM_CLOSE message by calling DestroyWindow to destroy the window. The DestroyWindow call sends WndProc a WM_DESTROY message, which WndProc processes.

The Dialog Box Procedures

Now is a good time to take a brief break from WndProc and examine the two dialog box procedures in CLOCK7.C. These are AlarmDlgProc, for the user to set the alarm, and AboutDlgProc, to display the program's About box.

The appearance of the dialog boxes is defined in dialog box templates in CLOCK7.RC. Each control in the dialog box is defined by a keyword, such as CTEXT for centered text and EDITTEXT for an edit control. The dialog box and each control have four numbers associated with them that indicate the placement of the upper left-hand corner of the window and the size of the window. These special dialog box units are based on the height and width of the default system font. Using these coordinates allows the dialog box to be approximately the same shape and size regardless of the resolution of the video display.

A dialog procedure is structured similarly to a window procedure, but with an important difference: whenever the dialog procedure processes a message, it returns TRUE. Otherwise, it returns FALSE. The dialog procedure does not call DefWindowProc.

AboutDlgProc is the simplest of the two dialog procedures. The dialog box contains some text and a pushbutton labeled "OK." When the user presses the pushbutton, it generates a WM_COMMAND message (just like a menu). The value of wParam is a number associated with the button in the dialog box template defined in the resource script. In this case, the button generates a wParam value of IDOK, an identifier defined in WINDOWS.H as 1. The dialog box also generates a WM_COMMAND message with wParam equal to IDCANCEL (the value 2) if the user presses the Escape key. In either case, AboutDlgProc calls EndDialog to dismiss the dialog box.

AlarmDlgProc is more complex. Besides some text, it contains five controls: a text entry field (for entering an alarm time), two radio buttons (labeled "On" and "Off"), and two pushbuttons ("OK" and "Cancel"). AlarmDlgProc performs some initialization during the WM_INITDIALOG message: the edit control is limited to six characters, and is initialized with a time stored in the szAlarmTime variable. One of the two radio buttons is set, depending on the value of bAlarmOn.

The WM_COMMAND message indicates that one of the two radio buttons or two pushbuttons has been pressed. For the two radio buttons, AlarmDlgProc stores the new state of the button. For the "OK" pushbutton, AlarmDlgProc obtains the time the user entered, parses it to obtain the hour and minute, and possibly displays a message box if the time is not valid. It ends the dialog box by calling EndDialog. For the "Cancel" button, AlarmDlgProc only ends the dialog box.

The WM_TIMER Message

Now let's return to WndProc to continue examining the messages.

The third message that WndProc processes is WM_TIMER. The WM_TIMER messages in CLOCK7 are initiated by a call to SetTimer during the WM_CREATE message. Windows posts a WM_TIMER message to CLOCK7's message queue once per second.

WM_TIMER processing begins with calls to the C time() and localtime() functions. If the bAlarmOn variable is set, and the time matches the time set by the user in the dialog box, CLOCK7 calls the MessageBox function to display a message box with the text "Wake Up! Wake Up! Wake Up!".

It then obtains a device context handle by calling GetDC, calls the DisplayTime function (described shortly) in CLOCK7.C, and releases the device context handle with a call to ReleaseDC.

The WM_PAINT Message

The first WM_PAINT message occurs during the UpdateWindow call in WinMain. This instructs the window procedure to paint its client area. Thereafter, a WM_PAINT message occurs whenever part of the window has become invalid and must be repainted. This could occur when the program is minimized and then redisplayed, or if part of the window has been obscured by another window and is then moved into full view.

CLOCK7 processes the WM_PAINT message similarly to the WM_TIMER message. It obtains the time by calling time() and localtime(), obtains a device context handle by a call to BeginPaint, updates the display with calls to DisplayTime and DisplayDots, and then ends WM_PAINT processing with a call to EndPaint.

The Drawing Functions

Toward the end of CLOCK7.C are the three functions the program uses to draw the clock: `DisplayTime`, `DrawDigit`, and `DisplayDots`.

`WndProc` calls the `DisplayTime` function while processing the `WM_TIMER` and `WM_PAINT` messages. The `DisplayTime` function calls `SetMapMode` to set the mapping mode for the device context, using the `MM_HIENGLISH` constant so that all coordinates passed to GDI functions are in units of 0.001 inch. Coordinates on the horizontal x axis increase to the right, and coordinates on the vertical y axis increase going up.

By default, however, the origin of the device context—the point (0,0)—is positioned at the upper right corner of the window. This is a little clumsy because all the coordinates within the window have negative y values. It is preferable to have the origin at the lower left corner of the window so that the window can be treated as if it were an upper right quadrant of a Cartesian coordinate system. The `SetWindowOrg` call in `DisplayTime` does this.

`DisplayTime` then calls `DrawDigit` six times, once for each of the six digits. The second and third parameters are the x and y coordinates of the lower left corner of the digit relative to the lower left corner of the window. For example, the lower left corner of the first digit is 100 units (1/10 inch) from the left side and bottom of the window. The last parameter is the digit to display. `DisplayTime` obtains these from the C `tm` structure containing the current time. If the first digit is zero, `DisplayTime` sets the parameter to 10, indicating a blank.

`DrawDigit` draws one digit on the window. Like a seven-segment LED display, each digit is composed of seven segments—three horizontal segments and four vertical segments. A number of variables are defined in `DrawDigit` to make the job a bit easier. The `bSegmentOn` array contains seven zeroes and ones for each of the digits 0 through 9 that indicate whether a particular segment should be illuminated or not (and 10, indicating the digit should be blank). The order of the seven segments is: the three horizontal segments from top to bottom, then the four vertical segments—top left, top right, bottom left, and bottom right.

The `iSegmentType` array contains seven ones and zeroes for the seven segments. A zero means that the segment is horizontal, and a one means the segment is vertical. The `ptSegOrigin` variable is an array of seven `POINT` structures that indicate the lower left corner of each segment, relative to the lower left corner of the digit. The `ptSegment` array contains the six points that define the outline of a horizontal segment, and the six points that define the outline of a vertical segment, all relative to the lower left corner of the segment.

DrawDigit begins by creating two brushes, a black brush and a red brush. These brushes are used to fill the segment. A segment that is illuminated must be colored red; a segment that is not illuminated must be colored black, the same color as the background of the window. (DrawDigit draws the black segments as well as the red segments to effectively erase the previous digit.)

The SaveDC function saves all the attributes of the device context. When all the drawing is complete, the RestoreDC call restores the saved attributes. This is necessary because DrawDigit alters the window origin attribute and must return it to normal in preparation for the next DrawDigit call.

When DisplayTime calls DrawDigit, the origin of the device context is at the lower left corner of the window. The OffsetWindowOrg function moves the origin to the lower left corner of the digit to be displayed, based on the second and third parameters to DrawDigit.

DrawDigit then loops through the seven segments. Once again, a call to SaveDC saves the attributes of the device context. The origin is then adjusted again for the particular segment based on the POINT structure in the ptSeg-Origin array. Now the origin is at the lower left corner of the segment in the digit. Based on the value of bSegmentOn, the SelectObject function selects either the red brush or the black brush in the device context. The Polygon function uses the six points in ptSegment to draw the segment and fill it with the brush. RestoreDC then restores the saved attributes of the device context (in particular, the device context origin) in preparation for the next segment.

The function cleans up by calling SelectObject to select the default black brush in the device context, RestoreDC to restore the device context to its default attributes, and DeleteObject to delete the two brushes.

The final drawing function in CLOCK7.C is DisplayDots, which displays the colon between the hour and minutes, and between the minutes and seconds. The function is called by WndProc only during processing of the WM_PAINT message. (It could be called from the WM_TIMER message, but it's not necessary because the dots never change position.)

DisplayDots creates a solid red brush and selects it into the device context. Four Ellipse functions draw the four dots. The second and third parameters to Ellipse are the x and y coordinates of the upper left corner of the dot; the fourth and fifth parameters are the x and y coordinates of the lower right corner.

DisplayDots cleans up by selecting the default black brush back into the device context, and deleting the red brush.

The WM_DESTROY Message

We have examined all the messages except WM_DESTROY. The WM_DESTROY message is the last message the window procedure receives. A window procedure usually takes this opportunity to do some clean-up work.

Examining in a little detail where the WM_DESTROY message comes from can be helpful in understanding how messages work, and the importance of DefWindowProc.

When you use the keyboard or mouse to select the Close option from the system menu, all keyboard and mouse activity takes place outside the client area. Window procedures usually ignore this activity because it involves messages that the window procedure can pass on to DefWindowProc for default processing.

When DefWindowProc determines that the user has selected Close from the system menu, it sends the window a WM_SYSCOMMAND message with wParam set to SC_CLOSE. WndProc ignores this message and passes it on to DefWindowProc. DefWindowProc responds by sending the window a WM_CLOSE message. WndProc ignores this message also and passes it on to DefWindowProc. DefWindowProc responds to the WM_CLOSE message by calling DestroyWindow. DestroyWindow destroys the window after sending it a WM_DESTROY message.

Why all this activity if WndProc is ignoring these messages? The window procedure doesn't have to ignore the messages. If it wants, it can intercept the WM_SYSCOMMAND or WM_CLOSE message in WndProc and prevent the program from terminating when the user selects Close from the system menu. The window procedure is being kept informed of what is going on even if it chooses to let DefWindowProc handle the messages.

The user can also exit the program by selecting the Exit option from CLOCK7's Set menu. As we saw earlier, this generates a WM_COMMAND message with wParam equal to IDM_EXIT, and WndProc responds by sending itself a WM_CLOSE message, just as DefWindowProc does when it receives a WM_SYSCOMMAND message with wParam set to SC_CLOSE.

WndProc responds to the WM_DESTROY message by cleaning up. The only clean-up required in CLOCK7 is to stop the timer by calling KillTimer. WndProc then calls PostQuitMessage. This function places a WM_QUIT message in the program's message queue.

When the GetMessage call in WinMain retrieves WM_QUIT from the message queue, GetMessage returns zero. This causes WinMain to drop out of the while loop and exit, terminating the program.

The CLOCK7.DEF Module Definition File

The only file we haven't discussed yet is CLOCK7.DEF, the module definition file. The module definition file contains information that LINK uses to create the CLOCK7.EXE executable. Most of the uppercase words in the CLOCK7.DEF file are keywords recognized by LINK.

The NAME line gives the program a module name, which is the same name as the program. The DESCRIPTION line is embedded in the CLOCK7.EXE file; this is generally a copyright notice. The EXETYPE is given as WINDOWS. (Module definition files are also used in OS/2 programming.)

The STUB line indicates a file of WINSTUB.EXE, a small DOS program included in the Windows Software Development Kit. The WINSTUB.EXE program simply displays the message *This program requires Microsoft Windows* and terminates. As mentioned earlier, the new executable format used for Windows programs begins with a header section that is the same as the MS-DOS .EXE format. The header could be followed by a non-Windows DOS program, which is specified in the WINSTUB statement.

The CODE statement in the CLOCK7.DEF file indicates that the program's code segment is MOVEABLE. This is normal. The DATA statement indicates that the program's data segment is also MOVEABLE. The MULTIPLE keyword means that each instance of the program gets its own data segment.

The HEAPSIZE statement specifies the size of the area in the program's data segment allocated for a local heap. This is really a minimum size, because Windows can expand the data segment if necessary. The STACKSIZE statement specifies the size of the stack, and 4K is normal for Windows programs.

Finally, the EXPORTS statement lists all the window procedures and dialog procedures in the CLOCK7 program. This is a requirement of Windows memory management. LINK stores the addresses of these functions in CLOCK7.EXE so that Windows can adjust the thunks to load the program's data segment address in the DS register when Windows calls the window procedure.

The Path to OS/2

Microsoft Windows forms the basis of the OS/2 Presentation Manager. Or maybe it doesn't. It depends on how you look at it.

Similarities and Differences

A Windows program certainly looks and feels a lot like a Presentation Manager program. It is one of the goals of Microsoft to maintain a consistent user interface between Windows and the Presentation Manager.

We have already mentioned several architectural components of Windows (namely, dynamic linking and the New Executable format) that have found their way into OS/2. Much of the overall architecture of Presentation Manager is based on Windows. The concepts of message procedures and messages are the same, and much of Presentation Manager windowing and user interface is based on Windows.

However, not one Presentation Manager function call is exactly the same as a corresponding Windows function call. At the very least, the function call name is different, and the parameters and data structures are often different as well. The Presentation Manager Graphics Programming Interface (GPI) is quite different from the Windows Graphics Device Interface (GDI).

Conversion from Windows to Presentation Manager

If you spend time writing a Windows program, eventually you will probably want to convert it to the OS/2 Presentation Manager. A real conversion requires that you go through the source files, sometimes on a line-by-line basis, and change the Windows code to corresponding Presentation Manager code.

Although it may seem as if this is equivalent to learning a whole new environment and completely rewriting the program, it's really not. If you already know Windows programming, learning Presentation Manager is not very difficult. Also, you've already spent a lot of effort in designing the user interface of the program, and this can be carried over into the Presentation Manager version. However, you may want to take advantage of some special OS/2 features (such as multiple threads), and this will require some restructuring of the code.

If you desire only that your Windows program run under Presentation Manager, that is not nearly as difficult. Micrografx has a product called Mirrors that is basically a collection of dynamic link libraries that translate Windows functions into Presentation Manager functions. It appears likely that OS/2 2.0 will allow running Windows applications directly in the Presentation Manager session using a similar technique.

In a longer time frame, you will be able to write single-source graphical programs that can be compiled for multiple platforms, such as Windows, Presentation Manager, and the Apple Macintosh. This will involve the use of an object-oriented language such as C++, and class libraries that provide a platform-independent interface to the particular graphical environment. At that time, programming directly to the Windows or Presentation Manager API will probably become as unusual as assembly language programming has become in recent years.

Chapter 7

DESQview

Stephen R. Davis

Like Windows, DESQview from Quarterdeck Office Systems is a DOS-compatible operating environment capable of executing one or several DOS programs simultaneously in a windowed environment.

DESQview can operate like a switcher by allowing the user to pop up a new DOS program at any time. The user first taps the "DESQ key" (usually defined as the Alt key, but user-selectable to anything desired) and then selects the application from the menu that appears. The new application appears immediately in a window over the old applications. The user may move or resize the window, either with the mouse or from the keyboard, to allow windows in the background to remain visible. He may also "zoom" the window out to fill the screen, completely hiding any background windows.

As with Windows, applications in the background continue to execute. A user may start a compilation, for example, and then pop up an editor to begin work on another source file in the foreground while the compilation completes. This is the famous "compiling in the background" capability.

DESQview runs as a shell over DOS. In this way, programs continue to "see" the familiar DOS interface, as much as possible. This keeps DESQview compatible with the vast majority of DOS applications. In fact, the DESQview-DOS combination looks enough like DOS that the user can bring up Windows within a

DESQview window (this is not true of Windows/386). Another advantage of this approach is that DESQview can hope to remain compatible with each new version of DOS. DESQview currently supports 2.0 through 4.x, and Quarterdeck is committed to supporting future versions as they are released.

DESQview adds capabilities to DOS beyond multitasking, for programs that care to use them. DESQview defines a series of service calls known as the Applications Program Interface (API). The API does not hide the basic DOS services, but serves as an extension of these capabilities. The API provides support for:

- character-based windowing
- multitasking
- pointers (such as the mouse)
- timers (both time duration and time of day)
- panels
- interprocess communication.

DESQview must have these capabilities for its own use. Through the API, DESQview makes these abilities available to application programs as well. DESQview even allows programs to control or limit capabilities normally provided through DESQview's pop-up menus. For example, a program can prevent the user from closing it prematurely from DESQview.

DESQview, as shipped from Quarterdeck, arrives on a single floppy disk within one spiral bound manual. The standard offering includes DESQview itself, a driver to load DESQview into memory above 640K, and a utility for loading drivers and TSRs into this high memory area to conserve conventional memory. Quarterdeck also offers, as a separate product, QEMM386, an EMS 4.0 emulator for 80386-based PCs. Although this driver may be used alone, it is designed specifically to be used with DESQview. Quarterdeck sells the two products together in a package known as DESQview386.

For the programmer interested in writing programs to access the DESQview API, Quarterdeck offers an assortment of tools. There are API Toolkits for C, Pascal, and Clipper, as well as the API Debugger, to aid in the development of such programs. There is also the Panel Design Tool, which allows the programmer to design windows the same way one might draw a picture with PC Paint; once complete, these windows are then incorporated directly into the user application.

The DESQview API

Broadly speaking, three types of programs execute under DESQview:

- *DESQview oblivious programs,* which are totally unaware of DESQview's presence
- *DESQview aware programs,* which are aware of DESQview's presence, but do not rely on it
- *DESQview specific programs,* which make use of DESQview's features through its API.

DESQview Oblivious Programs

DESQview oblivious programs are programs originally written for DOS. Since they know nothing of DESQview and multitasking, they do not expect other programs to be executing in the same machine simultaneously. Such programs expect to have total control of the computer, including all of its memory. As long as they are well behaved, DESQview has little trouble compelling them to coexist with their neighbors. If, however, they are "antisocial," there may be limitations.

The most common example of antisocial behavior occurs in the area of display output. In order to force the output of a normally full-screened program into a window, DESQview intercepts DOS and BIOS calls. Requests for video output are filtered and cropped into the appropriate window. However, to enhance video display speed, the majority of DOS programs do not output via DOS or the BIOS, preferring instead to write directly to screen memory. Being unaware of the physical boundaries of its window, such a program tends to write all over the display, including on top of the windows of other programs.

DESQview provides special programs called *loaders* for many of these offenders. These loaders are automatically executed when the user starts the program from the Open menu. The loader is given control after the program is loaded into memory but before it is allowed to start. The loader can then "reach into" the application and patch the output section so that its output may be windowed. Of course, in order to do this, the loader must have specific knowledge about the application. Each application must have its own unique loader.

On an 80386-based processor, DESQview can window screen-writing applications whether a loader exists or not. On these machines, DESQview uses the page mapping capabilities of the chip to allocate a different virtual display to each ill-behaved application. The application unknowingly writes to this virtual memory instead of the real video adapter. Periodically, a DESQview demon collects these virtual displays and paints them into the windows of the real video memory. This process is known as *virtualizing the application's display.*

However, if a loader does not exist for the program, and if the base processor is not an 80386 or better, then it is not possible for DESQview to window its output. Such programs can only execute in the foreground and only in full-screen mode. Zooming such an application into a window or switching it into the background suspends it.

A further problem with programs built to execute under DOS is that the .EXE file does not contain enough information for a multitasking environment. For example, DOS normally hands over all available memory to each new program as it begins. This is fine if you assume that no new program will start until this program has completed, which is true under DOS. But in a multiprogram environment, this assumption is not valid. In addition, if the program requires a loader, or must be virtualized in order to be windowed properly, this information must also be recorded.

To adapt DOS .EXE files to its own use, DESQview can use the same .PIF file used by Windows. Over the years, however, DESQview has added to the basic .PIF file, extending it into what Quarterdeck calls the .DVP (DesqView Pif) file. DESQview supplies .DVP files for the most common applications. These .DVP files can be created in the event one does not exist for a particular program, or edited to match the user's tastes at any time from the DESQview menu.

DESQview Aware Programs

The second category of software, DESQview aware programs, includes such bestsellers as Paradox, dBASE and WordPerfect. These programs acknowledge that they may not be the only task executing on the machine, and that they should refrain from dangerous or unfriendly tricks. For example, they use the BIOS clock instead of CPU timer loops to mark time, and they use the BIOS services rather than providing their own, as much as possible. However, such programs must still be able to write directly to screen memory. To allow these programs to execute within a DESQview window, DESQview provides shadowing capability.

The `api_shadow()` API call is built so that if DESQview is not present, DOS returns the call with no ill effect. If, however, DESQview is loaded, the system call returns the segment address of a logical display area to which the application can write. This allows displays to be virtualized on any CPU type.

DESQview Specific Programs

The final category of programs, DESQview specific programs, makes direct use of DESQview via the API. A DESQview specific program must first check that

DESQview is present and, if not, must terminate. The program can be set up so that DESQview is not apparent or accessible to the user (except for the copyright notice, which must appear when DESQview is initiated).

The DESQview API interfaces to C programs written by users through two .OBJ files included in the separate DESQview API C Library. (Separate libraries are also available for assembly language, Pascal, or Clipper programs.) Users must link these object files together with the objects from their own source files during the link step.

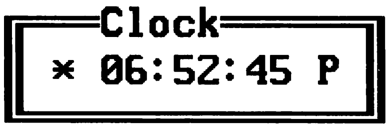
The API C Library defines 200 functions. In order to make the purpose of each function easier to remember, the functions are grouped into 12 categories, depending on what type of object the function accesses. The first three letters of a function's name indicate what category it belongs to. For example, functions beginning with "win_" act on windows; those beginning with "key_" read the keyboard; and those beginning with "ptr_" access the pointing device (mouse).

To better see how a programmer goes about writing a DESQview specific application, we can examine the same simple clock program that appeared in the discussion of Windows in Chapter 6, rewritten DESQview-style. Although a simple application, CLOCK is written in such as way as to make best use of the DESQview API. That is, CLOCK is constructed the way a much larger DESQview application would be. An examination of its features should teach the reader much about the DESQview API.

The Clock Example

Figure 7-1 shows a screen image of the CLOCK program executing. Since DESQview is not graphically oriented, it is difficult to generate displays such as analog clocks with sweeping second hands or the large "LED mimicking" windows typical of Windows. CLOCK's display is a single character-mode line encased within a window carrying the title "Clock."

Figure 7-1: The Clock window.

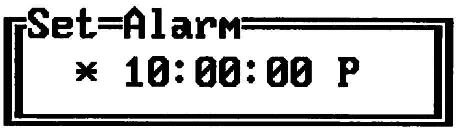


```
Clock
* 06:52:45 P
```

In the middle of the window is the time, which updates every second. Immediately to the right of the time is the AM/PM indicator. Selecting this field toggles the display between 12-hour and 24-hour mode. Since DESQview supports but does not assume the presence of a mouse, a field may be selected by either a) clicking on it, b) using the cursor keys to move the cursor over to it and pressing Enter, or c) pressing the "select key" for that field (in this case S for "switch").

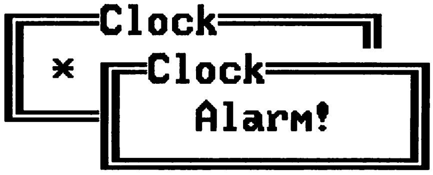
Just to the left of the time appears an asterisk. Selecting this field (either by clicking on it or by pressing the * key) opens the "Set Alarm" window shown in Figure 7-2. Although this window appears virtually identical to the first, the time (initially 00:00:00) does not update automatically. Instead this window waits for the user to enter an alarm time. Selecting the asterisk in this window closes the window, sets the alarm, and returns the user to the clock window.

Figure 7-2: The Set Alarm window.


```
Set=Alarm
  * 10:00:00 P
```

When the prescribed alarm time arrives, CLOCK opens a brightly colored Alarm window and beeps the speaker repeatedly at 2-second intervals (shown in Figure 7-3). Since CLOCK is designed to execute in the background, the Alarm window must automatically bring CLOCK to the foreground to ensure it is visible. Pressing Escape or clicking on the Alarm window removes the window and restores CLOCK to the background. The alarm may be disabled before it expires by reselecting the asterisk in the CLOCK window.

Figure 7-3: The Alarm window.


```
Clock
 *  Clock
      Alarm!
```

The source code for the DESQview specific CLOCK.C program appears on the following page:

```
/*CLOCK - this program displays a small clock on the screen.
         The user can position the clock anywhere on the screen
         desired using the DESQview menu. Display is in 12-hour
         format. Selecting the A/P indicator switches the
         clock to 24 hour format and back. Selecting the '*'
         button opens a new window to allow the user to set an
         alarm. The clock continues to run while the alarm is
         being set and when alarm window appears (clock update
         runs as its own task). The alarm may be disabled by
         deselecting the '*' button.*/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <string.h>
#include "dvapi.h"                   /*DESQview include file*/

/*prototypes for locally declared procedures*/
void main (void);
void program_body (void);
void alarmonoff (ulong panhan, ulong panelwin,
                 ulong timalarm, ulong panelkey);
void definealarm (ulong panhan);
void setalarm (ulong timalarm, struct timstruct *alarmtime);
void declarealarm (ulong panelwin);
int beep (void);
int ticktock (void);
void ourtime (struct timstruct *asciitime);
void bcd2asc (char *string, char bcdnum);
void updatetime (ulong panelwin, struct timstruct *asciitime);

/*minimum API version required is DESQview 2.00*/
#define required 0x200

/*panels containing the alarm and clock windows - built with
  the Panel Design Tool*/
extern char clockwin[];             /*this is the clock panel which
                                      contains the description of our
                                      clock windows*/
extern int lclockwin;               /*the length of the panel*/

                                    /*the clockwin panel defines the
                                      following fields*/
#define FLD_ALARM   01                   /*the alarm select field*/
#define FLD_HOURS   02                   /*hours*/
#define FLD_MINUTES 03                   /*minutes*/
#define FLD_SECONDS 04                   /*seconds*/
#define FLD_AMPM    05                   /*AM/PM indicator*/
```

```
/*global flags and structures*/
struct timstruct {              /*structure to hold current time*/
     char hour[3];
     char min[3];
     char sec[3];
     char afternoon;            /*'A' -> AM, 'P' - > PM*/
     };                         /*'*' -> 24 hour mode*/

struct timstruct currenttime;   /*currently displayed time*/
struct timstruct alarmtime =    /*last alarm time*/
           {"00", "00", "00", 'A'};   /*initial alarm time*/

struct panelmsg {               /*message from the panel*/
      char fldid;               /*field number*/
      int  length;              /*length of data*/
      char data;
      };

struct windowmsg {              /*message to ticktock subtask*/
      ulong windowhandle;       /*handle of window to write to*/
      };

int mode24;                     /*1 -> we are in 24 hour mode;
                                  0 -> we are in 12 hour mode*/

/*main - standard pattern for all DESQview programs -
         check to make sure that DESQview is present; if
         not generate error message; otherwise, proceed*/
void main (void)
{
  int  version;

  version = api_init();
  if (version < required)
       printf ("This program requires DESQview %d.%02d or later.\n",
                required >> 8, required & 0xff);
  else {
       /*tell DESQview what extensions to enable
         and then start real application*/
       api_level (required);
       program_body();
  }
  /*if DESQview present (even if wrong version), shut it down*/
  if (version)
       api_exit();
}

/*program body - open up the clock panel built with the Panel
```

```
                 Design Tool.  Define the necessary objects
                 and then sit in a loop waiting on messages*/

void program_body (void)
{
    ulong panhan;                 /*the handle for the CLOCKWIN panel*/
    ulong panelwin;               /*window handle for the clock*/
    ulong panelkey;               /*keyboard handle for clock panel -
                                    used to read select fields*/
    ulong timalarm;               /*timer handle used for alarm*/
    ulong tskhan;                 /*handle of clock update subtask*/
    ulong malhan;                 /*mailbox handle of subtask*/
    ulong obqhan;                 /*handle returned from object queue*/

#define STACKSIZE 1000
    char taskstack [STACKSIZE]; /*stack for the update time task -
                                  notice that this MUST be declared
                                  on a stack that is permanent*/
    struct windowmsg message;    /*message to send to subtask*/

    /*first order of battle is to open the panel file and display the
      clock panel - this will put a clock on the screen*/
    panhan = pan_new();
    if (!pan_open (panhan, clockwin, lclockwin))
        if (!pan_apply (panhan, win_me(), "CLOCK", 5,
                       &panelwin, &panelkey)) {

            /*start the ticktock task and send it a message with the
              window handle to start the clock ticking*/
            tskhan = tsk_new (ticktock, taskstack, STACKSIZE,
                              "", 0, 0, 0);
            malhan = mal_of (tskhan);
            message.windowhandle = panelwin;
            mal_write (malhan, (char *)&message, sizeof message);

            /*define the alarm clock timer*/
            timalarm = tim_new();

            /*now wait in an infinite loop for input from:
              - keyboard : user clicked a field; execute command
              - alarm    : alarm has gone off; put up alarm message*/
            for (;;) {
                /*now wait for an event to occur*/
                obqhan = obq_read();
                if (obqhan == panelkey)    /*keyboard input*/
                    alarmonoff (panhan, panelwin,
                                timalarm, panelkey);
                if (obqhan == timalarm)    /*alarm timer*/
                    declarealarm (panelwin);
            }
        }
}

/*alarmonoff - interpret the clock window's select field input*/
void alarmonoff (ulong panhan, ulong panelwin,
                 ulong timalarm, ulong panelkey)
{
     struct panelmsg *panelinput;
     int              inputlength;

     /*read the message from the clock's panel manager*/
     key_read (panelkey, &(char *)panelinput, &inputlength);

     /*field number 5 switches between 12 and 24 hour mode*/
     if (panelinput -> fldid == FLD_AMPM) {
          mode24 = (panelinput -> data == 'Y');

          /*update the clock in this task for instant response*/
          ourtime (&currenttime);
          updatetime (panelwin, &currenttime);
     }

     /*field number 1 sets/clears the alarm;
       N -> clear the alarm, Y -> set the alarm*/
     if (panelinput -> fldid == FLD_ALARM)
          if (panelinput -> data == 'N')
               tim_close (timalarm); /*stop the timer*/
          else {
               definealarm (panhan); /*set the alarm*/
               setalarm (timalarm, &alarmtime);
          }
}

/*definealarm - put up the alarm set panel where the user
                may enter the alarm time*/
void definealarm (ulong panhan)
{
     ulong alarmwin, alarmkey;
     struct panelmsg *alarminput;
     int              inputlength;

     /*first, open the ALARM panel to display Alarm Set window*/
     if (!pan_apply (panhan, win_me(), "ALARM", 5,
                               &alarmwin, &alarmkey)) {
```

```
          /*update the window to the previous alarm time...*/
          updatetime (alarmwin, &alarmtime);

          /*...and position the cursor for time entry*/
          fld_cursor (alarmwin, FLD_HOURS);

          /*now wait for the user to update the time fields*/
          for (;;) {
               key_read (alarmkey, &(char *)alarminput, &inputlength);
               switch (alarminput -> fldid) {

               /*selecting fields 2 thru 5 just
                 fills values into the alarm time*/
               case FLD_HOURS:
                   strncpy (alarmtime.hour, &alarminput -> data, 2);
                   break;
               case FLD_MINUTES:
                   strncpy (alarmtime.min,  &alarminput -> data, 2);
                   break;
               case FLD_SECONDS:
                   strncpy (alarmtime.sec,  &alarminput -> data, 2);
                   break;
               case FLD_AMPM:
                   alarmtime.afternoon = alarminput -> data;
                   break;

               /*selecting field 1 removes the alarm window and
                 returns control to the main program*/
               case FLD_ALARM:
                   win_free (alarmwin);
                   return;
               }
          }
     }
}

/*setalarm - start the alarm timer*/
void setalarm (ulong timalarm, struct timstruct *alarmtime)
{
     ulong settime;
     int hours, mins, secs;

     /*convert the hours, minutes, and seconds into time since
       midnight*/
     hours = atoi (alarmtime -> hour);
     mins  = atoi (alarmtime -> min);
     secs  = atoi (alarmtime -> sec);
     /*if PM, add 12 to the hour (don't make result > 24)*/
     if (alarmtime -> afternoon == 'P')
          hours += 12;
     hours %= 24;

     /*convert the entire thing into 1/100ths of seconds since
       midnight and set the alarm to go off at that time*/
     settime = ((((hours * 60L) + mins) * 60L) + secs) * 100L;
     tim_write (timalarm, settime);
}

/*declarealarm - alarm has gone off; open an alarm window and
                 display alarm message*/
void declarealarm (ulong panelwin)
{
    ulong beeptaskhan;
    char  stack [400];

    char *msg = "   Alarm!   ";
    int  length;

    /*push ourselves into the foreground so user can see us*/
    app_gofore (win_me ());

    /*start a subtask beeping every 2 seconds*/
    beeptaskhan = tsk_new (beep, stack, sizeof stack, "", 0, 0, 0);

    /*display a simple alarm message - user acknowledges with Escape
      or by clicking on it with the mouse*/
    length = strlen (msg) - 1;
    win_disperor (panelwin, msg, length, 1, length, 0, 0);

    /*stop that infernal racket - kill the beeping task and
      unqueue any sound*/
    tsk_free (beeptaskhan);
    api_sound (0, 0);

    /*now push ourselves back into the background...*/
    app_goback (win_me ());

    /*...and deselect the alarm set button to show no alarm pending*/
    fld_type (panelwin, FLD_ALARM, FLT_DESELECT);
}

/*beep - a subtask to beep every two seconds when alarm goes off*/
int beep (void)
{
    for (;;) {
            api_sound (1000,  9);  /*1000 Hz for  .5 (9/18.2) sec*/
            api_sound (   0, 27);  /*silence for 1.5 secs*/
        }
}


/*ticktock - small subtask used to constantly update the clock*/
int ticktock (void)
{
    ulong winhandle;            /*handle of window to update*/
    ulong timpause;             /*handle of timer to delay with*/
    struct windowmsg *winmessage;
    int              messagelength;

    /*first read the message sent to us with the handle of the
      window to which we should write the time*/
    mal_read (mal_me(), &(char *)winmessage, &messagelength);

    /*save the clock window handle locally*/
    winhandle = winmessage -> windowhandle;

    /*now define an update timer*/
    timpause = tim_new();

    /*sit in a loop - */
    for (;;) {
        /* - start the timer for 1 second and wait for it to expire*/
        tim_addto (timpause, 100);
        tim_read (timpause);

        /* - when it does, (re)display the time*/
        ourtime (&currenttime);
        updatetime (winhandle, &currenttime);
    }
}

/*ourtime - get the current time into an ASCII structure*/
void ourtime (struct timstruct *asciitime)
{
     union REGS regs;

     /*use the BIOS Get-Time-of-Day call to quickly get the time*/
     regs.h.ah = 0x02;
     int86 (0x1A, &regs, &regs);

     /*set afternoon flag and round the hour off if in 12 hour mode
       (notice that arithmetic on BCD is slightly strange)*/
     asciitime -> afternoon = '*';
     if (!mode24) {
         asciitime -> afternoon = 'A';
         if (regs.h.ch > 0x12) {
              asciitime -> afternoon = 'P';
             if (((regs.h.ch -= 0x12) & 0xf) > 0x09)
                 regs.h.ch -= 6;
         }
     }

     /*convert the BCD to ASCII strings*/
     bcd2asc (asciitime -> hour, regs.h.ch);
     bcd2asc (asciitime -> min , regs.h.cl);
     bcd2asc (asciitime -> sec , regs.h.dh);
}

/*bcd2asc - convert a BCD number into an ASCII string*/
void bcd2asc (char *string, char bcdnum)
{
     *string++ = (bcdnum >> 4)   + '0'; /*upper digit*/
     *string++ = (bcdnum & 0x0f) + '0'; /*now the lower digit*/
     *string   = '\0';
}

/*updatetime - update the time display by writing the time to
               the hours, minutes, second and AM/PM fields of
               the specified panel window*/
void updatetime (ulong panelwin, struct timstruct *asciitime)
{
    /*write the time into fields 2 thru 4*/
    fld_write (panelwin, FLD_HOURS,   asciitime -> hour, 2);
    fld_write (panelwin, FLD_MINUTES, asciitime -> min,  2);
    fld_write (panelwin, FLD_SECONDS, asciitime -> sec,  2);

    /*and put up the AM/PM indicator into field 5*/
    fld_write (panelwin, FLD_AMPM,   &asciitime -> afternoon, 1);
}
```

The first attribute that distinguishes this from a non-DESQview specific program is the appearance of: `#include"dvapi.h"`. This include file defines the constants commonly used in API calls. It also contains the prototype definitions for the API functions. Two versions of the file are supplied with the API C Library: one that supports the Kernighan and Ritchie standards and another that supports the more stringent ANSI C typing standards (available in versions 1.1 and later of the C Library).

The API interface functions are contained in two object files also supplied with the API C Library: API1.OBJ and API2.OBJ, which must be linked with the user's object files. Versions for the major C compilers are included, but adapting to other C compilers should be simple, since the source code to both object files is provided. Linking these object files in can be handled either specifically, as with this Microsoft C example:

```
        LINK USER.OBJ+API1.OBJ+API2.OBJ,,/ST:32768
```

or, in languages that support MAKE or Project files as in this Turbo C project file:

```
            API1.OBJ API2.OBJ
            USER.C
            LIB\CL.LIB
```

These object files are quite small, since they only provide the interface between the user's high level language and the API, which is a permanent part of DESQview and always present. The API itself remains a part of the operating system and not the application.

All pointers in the DESQview API are far pointers (32-bit). It would be possible to carefully define all pointers to be of a far type; however, the functions provided in the API libraries carry far addresses as well. Therefore, Quarterdeck strongly advises all developers programming for the API to compile only under the large memory model. Most C compilers allow compiling under the large memory model. Under Turbo C, select Large under the Options/Compiler/Model menu and specify the Large C library in the project file, as in the example above. Under Microsoft C, use the /AL switch.

In addition, structures under DESQview are assumed to be packed on byte boundaries. Any module that exchanges a structure with the API should specify byte alignment. Under Turbo C, select Byte alignment from the Options/Compiler/Code Generation menu. Under Microsoft, specify /Zp on the compile line. (If your compiler does not support byte alignment, you can get around this by declaring all structure members to be integers, and packing the bytes yourself.)

The `main()` for CLOCK follows the pattern for all DESQview-specific programs:

```
#include "dvapi.h"

/*main - standard pattern for all DESQview programs -
         check to make sure that DESQview is present; if
         not, generate error message; otherwise, proceed*/
void main (void)
{
  int  version;
```

```
     version = api_init();
     if (version < required)
         printf ("This program requires DESQview %d.%02d or later.\n",
                    required >> 8, required & 0xff);
     else {
          /*tell DESQview what extensions to enable
            and then start real application*/
          api_level (required);
          program_body();
     }
     /*if DESQview present (even if wrong version), shut it down*/
     if (version)
          api_exit();
}

/*program body - do the actual work here*/
void program_body (void)
{
    ...
}
```

Since DESQview does not have an .EXE format that distinguishes DESQview specific programs from others, CLOCK must first check to make sure that DESQview is present before continuing. Even if DESQview is present, CLOCK should make sure that it is a current enough version to support the functions used within the program. CLOCK requires version 2.00 or better, and so tests for this. The `api_init()` service call is designed to return the version number if the program is executing under DESQview and return a 0 if it is not.

Windowing

Generally, the first thing a program wants to do upon starting is open a window on the screen. Sometimes this is not necessary, since DESQview automatically opens the first window when the program begins. But even then, the application cannot be sure how large the window is, where it is placed or what the default color scheme might be (these things are specified in the .DVP file, which is subject to change by the user). Therefore, even if the application uses the default window, it will probably want to place, size, and color the window itself.

Thus, a typical program begins as follows:

```
ulong windowhandle;
char windowname[] = {"Main Process"};

/*Open initial window, then position, size and draw it*/
```

```
windowhandle = win_new(windowname, sizeof windowname - 1, 10, 10);
win_move (windowhandle, 1, 1);
win_attr (windowhandle, 1);
win_unhide (windowhandle);
win_redraw (windowhandle);
```

The first call, that to `win_new()`, creates a window bearing the name "Main Process" in the upper left corner of the border. This window has a border (the default), and is 10 columns wide by 10 rows high. Since the DESQview API is not C-based and does not subscribe to the "null terminates strings" rule, it is necessary to give the length of the window's name (minus the null on the end) as well as the name itself (this is true of all strings passed to the API).

DESQview is designed around character mode. This is why the window size information is based on columns and rows and not on pixels. The box that outlines a window is constructed using the special *block graphics* characters. A window may contain graphics information, requiring the video adapter to be in a graphics mode; however, even in graphics mode, DESQview draws and positions its character-based windows.

Notice that what is returned from the `win_new()` call is a `long int` (`ulong` is defined in DVAPI.H as an `unsigned long int`). This identifier is not unlike the file handles with which C manipulates files. The long `int` returned from `win_new()` will be used by the remainder of the program to identify that window. Since a program may have several windows open at time, these identifiers serve to keep them straight. In DESQview nomenclature, the new window is an *object,* and the identifier is the *handle* of that object (hence the variable name `windowhandle`).

As we will see, handles exist for other types of objects. For example, keyboard input is handled through a keyboard object. Each type of object is manipulated by its own type of API function. To distinguish the various types of API calls, the first three letters of the function name refer to the type of object the function manipulates. Thus, `win_` calls are for window objects and `key_` calls for keyboard objects. We will see several other types of objects as we proceed. An object handle *may* be a pointer to a structure DESQview uses to describe the object, but the programmer should simply think of it as a *tag* or *identifier.* A program should *never* try to access the object directly by using the handle as an address.

The next window call, that to `win_move()`, positions our new window at row 1, column 1; that is, the upper left-hand corner of the display. The `win_attr()` call selects the color scheme. Up to 16 different color schemes may be defined, and these may be defined differently for CGA, monochrome, and EGA/VGA adapters, to best utilize the colors and shades available for each screen.

The final two calls, `win_unhide()` and `win_redraw()` are interesting. All windows are created "hidden" (the default window created when the application was first started is unhidden by DESQview before the application is given control). A hidden window is invisible. Even making it unhidden, however, is not sufficient to make it appear on the display. Calls to `win_` functions change the window object in memory, but not on the screen. It is not until the `win_redraw()` API call that the screen is actually "redrawn" onto the display. Although this may seem odd, there are several good reasons for this.

First, just as an architect does not generally want the public to see the frame holding up a building, the programmer does not want the user to see all the window manipulations going on behind the scenes to make the final display. If the window were visible as soon as it was created in our example, the user would first see the window created in some unknown location, then redrawn in the upper left-hand corner, but with the wrong color pattern, and finally redrawn yet again with the correct color scheme. It is better that these window manipulations go on behind the scenes so that the user only sees the window appear once, after it is ready.

Another reason concerns execution speed. Video memory is generally very slow, since the CPU must compete with the video adapter for access. Redrawing the window each time before it is actually ready wastes time. It is faster to make all the adjustments before writing the window to display memory a single time.

A third reason is related to response times. Consider, for a moment, a program with three different drop-down menus used for different command types. Each of these menus would normally be implemented as a separate window. The program could wait until the user clicked onto a menu bar option before creating the menu window and unhiding it. If we assume that the user will click on each menu bar option at least once, a more attractive option presents itself. The application can create all of the menu windows when it is first started. When the user decides to select a menu bar option, all the application must do to make it visible is to unhide and redraw it. When the user is finished with a menu, the program hides it again, where it remains ready for the next time it is needed. Although building all menus at once slows down program initialization a bit, this makes the program respond very rapidly to user input. Users expect programs to take a certain amount of time to start, but they like to see commands executed quickly.

The only major win_ API calls that do not require win_redraw() are the printf-type functions, such as win_printf().

The window that is automatically opened for the application, sometimes called the *default window*, is initially sized and positioned by information in the .DVP used to open the application. This window may be manipulated the same as any other window. Its handle is returned from the win_me() API call. For example, to resize its initial window, an application might do the following:

```
ulong defaultwindow;

defaultwindow = win_me();
win_resize (defaultwindow, num_rows, num_cols);
win_redraw (defaultwindow);
```

We should note at this point that windows opened by DESQview are different than those opened by add-on windowing libraries. As we have already seen, the code to manipulate DESQview windows is part of the environment, and not part of the application. Although this makes the application dependent on DESQview, it also means that the application can be considerably smaller.

Even more important, applications can do much more with DESQview windows. For example, a window opened with a win_new() call can be manipulated by the user using the normal Resize and Move menu commands. (The application may prevent this from happening with the win_disallow() API call.) Further, an application may push a window in front of or behind other windows, even if they belong to other applications. In our sample program, since CLOCK does not specifically forbid it, the user can put the clock window anywhere on the screen.

The concept of handles associated with objects is an important one in DESQview. DESQview relies heavily on them. For example, all input and output under DESQview is through object handles. This is why both a window object and a keyboard object must be opened automatically at program initiation: DESQview converts calls to BIOS screen and keyboard routines into win_ calls to the default window object, and key_ calls to the default keyboard object. In this way, a DESQview oblivious program that performs a printf() actually performs a win_printf(win_me()) in a roundabout way.

Panels

A DESQview application tends to have many windows: a window for each different menu, a window for different types of data, and so on. It is possible for the application to create and manipulate each of these windows using discrete `win_` API calls. However, this might result in a confusingly large number of `win_`calls to set things up properly, and changing a menu or output window might prove difficult. Quarterdeck has made window generation much easier by providing a separate utility known as the Panel Design Tool.

The Panel Design Tool allows the programmer to "draw" windows using a MacDraw or PC Paint type of interface. The programmer draws out a box, sets such properties as border type and color from menu bars across the top, and types in any text that should appear within the window. DESQview calls a window generated in this way a *panel*. Each panel is given an eight-character name so that it may be identified easily. Once they are completed, the Panel Design Tool combines up to 255 panels into what is known as a *panel file*.

Panel files may have either of two formats: they may be a separate binary file that the program reads at execution time, or they may take the form of a .OBJ file, which the programmer simply links in with the application at link time. The latter method is usually more convenient.

Both the Clock and Set Alarm windows within CLOCK are implemented as panels within a Panel File known as CLOCKWIN. The CLOCKWIN.OBJ generated by the Panel Design Tool must be linked together with CLOCK.OBJ, just like the API1.OBJ and API2.OBJ files. CLOCKWIN.OBJ defines two global labels that CLOCK needs: `clockwin` and `lclockwin`. `clockwin` is the address of the panel file in memory and `lclockwin` is its length.

To make a panel window appear on the screen, the program must first acquire the handle of an empty panel object using the `pan_new()` call. Opening the panel file using the `pan_open()` API call makes the individual panels available for display. Applying the panel using the `pan_apply()` API call displays the window. In CLOCK it looks like:

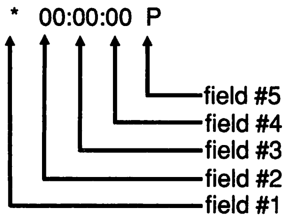
```
extern char clockwin[];          /*this is the clock panel that
                                   contains the description of our
                                   clock windows*/
extern int lclockwin;            /*the length of the panel*/

ulong panhan;                    /*the handle for the CLOCKWIN panel*/
ulong panelwin;                  /*window handle for the clock*/
ulong panelkey;                  /*keyboard handle for clock panel -
                                   used to read select fields*/

panhan = pan_new();
if (!pan_open (panhan, clockwin, lclockwin))
```

```
        if (!pan_apply (panhan, win_me(), "CLOCK", 5,
                        &panelwin, &panelkey))
```

Notice that if either the pan_open() or the pan_apply() function returns a non-zero value, the function was not successful and the program should terminate (the value returned is indicative of the problem encountered). Also notice that pan_apply() returns the handle of a window object. The window opened by a panel may be further manipulated just like any other window, using this handle. In addition, pan_apply() returns the handle of the keyboard object that the application should read to receive input from the window.

You cannot only paint fixed text into windows using the Panel Design Tool, but also define variable text areas known as *fields*. Each field within a panel window carries a number. You can output to these fields via their field numbers. This is the way CLOCK handles output. In both the Clock and Alarm Set windows, the fields are numbered as in Figure 7-4.

Figure 7-4: Field definitions for Clock and Set Alarm panels.



Output to these fields is via the fld_write() API call. #defines can be used to make the meaning of the different fields clearer. In the case of CLOCK, these defines look like this:

```
                                           /*the clockwin panel defines the
                                                following fields*/
    #define FLD_ALARM    01                /*the alarm select field*/
    #define FLD_HOURS    02                /*hours*/
    #define FLD_MINUTES  03                /*minutes*/
    #define FLD_SECONDS  04                /*seconds*/
    #define FLD_AMPM     05                /*AM/PM indicator*/
```

For example, when it is time to display the current time from the structure asciitime into a window, CLOCK uses the following function:

```
    /*updatetime - update the time display by writing the time to
                   the hours, minutes, seconds and AM/PM fields of
```

```
                  the specified panel window*/
   void updatetime (ulong panelwin, struct timstruct *asciitime)
   {
      /*write the time into fields 2 thru 4*/
      fld_write (panelwin, FLD_HOURS,   asciitime -> hour, 2);
      fld_write (panelwin, FLD_MINUTES, asciitime -> min,  2);
      fld_write (panelwin, FLD_SECONDS, asciitime -> sec,  2);
      /*and put up the AM/PM indicator in field 5*/
      fld_write (panelwin, FLD_AMPM, &asciitime -> afternoon, 1);
   }
```

The final argument to `fld_write()` is simply the length of the string supplied in the third argument.

Fields give panels their true power. Panels are more than just a method to define windows quickly without coding many series of boring `win_` calls. Panels allow you to separate the form of an application from its function. When CLOCK outputs the time to fields 2 through 5, it does not know or care where they are in the window. If you wanted to move (or remove) fields within the Clock or Set Alarm windows, you would simply redraw the screen using the Panel Design Tool, and relink. It would not be necessary to change the C program at all.

The CLOCK program is probably too simple to seriously consider going to the trouble; however, large, real-world applications change continuously. New windows may be added, requiring existing windows to be redesigned in order to make room. Changing existing source code to accommodate such format changes requires programmer time plus expensive retesting, and invites errors. Panels allow you to restructure the output of a program using the Panel Design Tool without the need to change the application code, resulting in considerable savings in cost and time.

Another problem addressed by panels is that of multiple versions. Some companies offer several versions of the same program. This may be to support different human languages, different price tags, or different levels of user sophistication (for example, while the instructor version of some test administering software might have a "Show Correct Answer" menu option, the student version most certainly would not).

While generating modified versions of software from the original is not conceptually difficult, it is a process full of pitfalls. Adding conditional compilation statements or commenting out sections of code invites programming problems. The Panel Design Tool allows you to code an application once. You can then create different versions of the application by changing the panel file. Reduced versions use panels with fields missing or replaced. Thus, in our example above, the student can't "Show Correct Answer," not because the code won't allow it, but because the panel does not display the option to do so. Of course, the entire panel may be rearranged so that it is not obvious to the student that the missing field was ever present.

DESQview Tasks

Of course, the DESQview API also supports multitasking. Before discussing multitasking, we need to define two very important terms: *task* and *process*. A *task* in DESQview is a single-execution thread; that is, a path the CPU takes through the program. A related concept is that of *process*. A process is the total of all memory areas unique to a program. Thus, a single-tasked application consists of a single task (one execution thread) within a single process (the application's code, data, and window memory).

A simple analogy is a wood with several trails. The process corresponds to the wood itself: the trees and the area they cover. There are multiple trails that people may take through the woods, just as there are many logical paths through an application. A task corresponds to a hiker strolling along one of these trails.

It is not always clear when multitasking is useful. CLOCK could be implemented in a single task; however, it would be difficult to accept user input, including setting the alarm time, and still update the time reliably every second. Therefore, CLOCK consists of two tasks: one to accept user input and a second independent task that simply puts up the time every second. Even when the first task is blocked, awaiting input, the second task ticks reliably on.

Starting a new task in DESQview is handled with the `tsk_new()` API call shown below. The parent task must supply the address of the function where the subtask is to begin, the location of its stack, and the name and size of its initial window. In this example, the subtask begins with the function `ticktock()`. It is allocated `STACKSIZE` bytes of stack, and given a window of size `NUMBER_ROWS` by `NUMBER_COLS`, bearing the label "Subtask." The value returned from `tsk_new()` is the task handle that is used to identify the task in future API calls.

```
ulong taskhandle;
int ticktock (void);
char stack [STACKSIZE];
char windowname[] = {"Subtask"};

taskhandle = tsk_new (ticktock, stack, STACKSIZE,
                      windowname, sizeof windowname - 1,
                      NUMBER_ROWS, NUMBER_COLS);
```

Notice that a task's stack must be a section of memory that will remain untouched by any other task. It may be returned from a `malloc()` call or defined as a fixed global array; however, any stack checking code that may be inserted by the compiler may have difficulty with a stack segment that is different from the main task's. Most notably, Microsoft C has a problem in this area. This can be addressed by defining the subtask's stack within the main task's stack (as CLOCK does). In this case, it must be declared in a function that *does not* return as long as the subtask is executing, since returning would surrender the memory space for other uses.

If a task is to operate silently in the background, it may not need a window. This is also true if the task intends to operate on a window that has already been opened by another task. In either case, you may specify a window size of zero rows by zero columns and no window will be opened on the screen when the task is started. This is the case with the `ticktock()` function in CLOCK.

Invoking a subtask via `tsk_new()` is very similar to calling a function, except that control returns to the caller before the subtask completes. If we look at our "hiker in the woods" model, calling a normal function is similar to a hiker taking a side path off the main trail. This side path winds around, perhaps taking side paths of its own, before eventually returning to the main trail via the function return. No matter what happens in the side path, progress down the main trail is suspended until the hiker returns from the side path. Spawning a subtask is as if the hiker splits in two. The original hiker is allowed to continue down the main trail while the clone simultaneously sets off along the side path.

Tasks can terminate in one of three ways. If the program aborts for any reason, all of its tasks are automatically terminated. Alternatively, if a subtask attempts to return from its highest-level function (the one named in the `tsk_new()` call), DESQview terminates it. Conceptually, this is identical to a single-tasked program returning from `main()` to the operating system. Finally, a task may be aborted at any time via the `tsk_free()` call. The `tsk_free()` call requires the task handle returned from the `tsk_new()` call that created it. Freeing a task's object kills the task. (A task may acquire its own handle using the `tsk_me()` API call if it needs to kill itself.)

Of course, there is only one CPU, so the two tasks don't actually make progress simultaneously. One task proceeds a few steps along the program trail, or until the next time it must wait for input, and then the second task gets a turn to proceed. Still, it is easier for the programmer to train several hikers individually, each in a single job, than to train one hiker task to do it all.

DESQview uses a preemptive tasking algorithm to decide which task gets control of the CPU at any given time. That is, scheduling is based on the hardware clock. Each task is allowed to execute for a certain number of clock ticks—the task's time slice—before control is wrestled from it and given to the next task in line.

A task normally executes for a complete time slice, but it may give up the remainder of its time slice if it runs out of things to do. For example, a program suspends itself any time it reads the keyboard. Suspended tasks do not receive any CPU time. A task may also have its time slice taken away from it if it polls the keyboard more than a given number of times in a single time slice (this time is specified in the .DVP file). A DESQview specific program can pass control on without suspending itself using the `api_pause()` API call.

DESQview recognizes three types of tasks: the foreground task, the background tasks, and interrupt handlers. All background tasks are given the same time slice. The foreground task is given a different time slice. The defaults are nine clock ticks for the foreground task and three for each background task, but the user can change these both at Setup and from the DESQview menu. Giving the foreground task more time results in livelier keyboard response at the expense of background tasks. Foreground and background tasks are scheduled in a "round-robin" fashion. DESQview may suspend a foreground or background task to schedule an interrupt task immediately to service a hardware interrupt.

As we noted already, tasks within the same process share the same code and data space. Such tasks appear as functions within a single executable file and are bound together at link time. This must be so, since the argument to `tsk_new()` is the address of the function that is to begin the subtask. Thus, tasks "know" about each other. They have access to all the same functions and all the same global variables. Both of these facts present their own problems.

Access to common global variables brings with it certain problems in a multitasking environment. Since the programmer cannot be sure when the hardware alarm will strike and the task will lose control, a task cannot write to a location that another task may write to, and expect the value to stay unchanged. This is true whether the location is in RAM or on disk.

To illustrate, suppose that your bank uses an accounting program with two tasks, task A and task B, both of which use a global variable `customer_balance`. Suppose also that task A decides it is time to credit the daily interest to your savings account. Task A dutifully loads your balance into the global variable and calculates your new balance. Before it can write this balance back out to disk, however, task A's time slice is up and control is passed to task B. As luck would have it, task B's job is to credit a deposit that has just arrived. Task B reads some other person's account into `customer_balance`, adds the deposit, and writes the result back out to disk. When task A eventually gets control back, it continues with its write operation, writing the contents of `customer_balance` into your savings account. Unfortunately for both you and the bank, the content of `custo-mer_balance` now has nothing to do with the real balance in the savings account.

The problem here is that task A and task B use the same fixed memory location, `customer_balance`. Solving the problem in this case is quite simple. There is no reason to locate `customer_balance` in global memory. If both task A and task B are modified to declare `customer_balance` locally, the conflict does not occur. While both tasks might continue to refer to a variable `customer_balance`, since it is now declared locally, it no longer refers to the same location in memory.

A similar problem arises with functions shared by subtasks. Suppose, for example, that the global variable in the above example is only accessed from a single function, say `update_balance()`. We might think that no conflict is possible; that is not the case, however, since `update_balance()` might be executed by both task A and task B at the same time. To return to our hiker analogy briefly, `up-date_balance()` is simply a stretch of trail that two hikers might walk along at the same time. In this case, we say that the function `update_balance()` is not reentrant; that is, it cannot be reentered while another task is in the function.

It is interesting to note that DOS itself is not reentrant; two different tasks may not execute DOS service calls at the same time. DESQview must be careful to control tasks entering and exiting DOS to preclude this from happening. (It is widely known that DOS actually consists of two parts: calls with numbers below `0BH`, and those above. These two sections are mutually reentrant, in that one program may be executing one of the lower DOS calls while another is executing a call from the upper range.)

If a task attempts to make a DOS call and DESQview detects that a second task is already in the middle of such a call, the calling task must be suspended. When the second task is subsequently given control, it will eventually complete its DOS call. As it exits DOS, DESQview unsuspends the first task, allowing it to continue into DOS the next time it is scheduled.

The ANSI.SYS device driver provided with DOS is not reentrant either. In this case the solution is much simpler, however: DESQview comes with a reentrant version of ANSI.SYS called ANSI.COM, which can be installed within an application window.

As an aside, tasks that are part of the same executable file share a single process and, therefore, share the same .DVP file. It is not possible to specify that one such task writes directly to screen memory and should be virtualized, while another task does not.

In addition, the user controls multiple tasks of the same program together. Selecting a window of one task brings the windows of all tasks that are part of the same program to the foreground together. It is not possible for the user to suspend one task without affecting the rest. If, for example, a program opens a menu controlled by another task, the user should not be able to select the menu separately from the program that created it. As far as the user is concerned, all task windows are part of the same program.

DESQview does allow a type of parent task that can be handled individually. DESQview calls it an *application*. A normal task belongs to the parent task which created it. Acting on the parent task (by selecting it or closing it, for example) affects all of its subtasks. A newly spawned application task doesn't belong to a parent task. It becomes its own parent and can be operated on independently.

Take the example of a spreadsheet program. All the drop-down menus are handled by normal subtasks since they are part of the application. Suppose, however, that the program allowed multiple spreadsheets to be worked on at one time. This would best be handled by spawning the same code as separate applications, one for each spreadsheet. This would allow the user to manipulate the spreadsheets separately. The user could bring one spreadsheet to the foreground, leaving another in the background; zoom a spreadsheet on top of the other windows in the background; even close a spreadsheet from the DESQview menu.

A new application is spawned using the `app_new()` API call, whose arguments are identical to those of `tsk_new()`. The `app_new()` call returns a task handle just like that returned from `tsk_new()`. The user has the illusion that the applications are completely independent programs, even though they share the same code and data space. The only exception to this rule is that if DESQview must write an application to disk, it must write all applications within the same process to disk. It is the process that the applications share which actually gets written.

DESQview Processes

Just as a task can spawn a subtask in the same process, so can a task spawn a subtask in a different process. This is analogous to the DOS or UNIX `exec` call, in which one program can execute another program. DESQview handles this using the `app_start()` API call. Unlike the DOS `exec()` command, the argument to `app_start()` is the address of the .DVP structure for the process.

```
ulong taskhandle;
struct DVP processDVP;

taskhandle = app_start (&processDVP,
                        sizeof processDVP);
```

A task created by the `app_start()` call is completely independent from the task that started it. The two tasks share neither code nor data. Their windows are independent—one may be brought to the foreground and the other left in the background. The two tasks may be swapped out to disk or terminated independently. Because they occupy different processes, the two tasks are like completely separate programs.

The `ticktock()` function could be written as a separate process. To do so, we would create a separate TIKTOK.C file containing a `main()` that first checked for the presence of DESQview and then called `ticktock()`. Any functions that `ticktock()` shared with the rest of CLOCK would be repeated in TIKTOK. We would then compile CLOCK, generating a CLOCK.EXE without `ticktock()` and a TIKTOK.EXE with only `ticktock()`. When CLOCK performed the `app_start()` call, DESQview would load TIKTOK.EXE into a separate memory area and start it.

The .DVP structure provided to `app_start()` may be initialized by reading a .DVP file into a buffer from the disk. Alternatively, the necessary data may be hard-coded in the source program using the C structure that appears in the example below. It is not necessary to supply a .DVP structure when starting a new task, since each subtask remains in the same process. With `app_start()`, however, we are not only creating a new hiker, we are building a whole new wood. Notice that it is not necessary to specify the name of the executable file to `app_start()` since this is contained in the .DVP structure along with the rest of the information.

```
struct DVP {
  unsigned reserved1;         /*set to 0x0097*/
  char     title [30];        /*program title - pad with blanks*/
  unsigned maxmem;            /*maximum memory required [kbytes]*/
  unsigned minmem;            /*minimum memory required [kbytes]*/
  char     command [64];      /*start command [ASCIIZ]*/
  char     drive;             /*default drive in ASCII
```

```
                                 (blank for none)*/
char     directory [64];        /*default directory [ASCIIZ]*/
char     params    [64];        /*parameters to command [ASCIIZ]*/
char     screenmode;            /*initial screen mode (0 - 7)*/
char     numpages;              /*number of video pages*/
char     firstvect;             /*first interrupt vector to be saved*/
char     lastvect;              /*last interrupt vector to be saved*/
char     rows;                  /*no. rows in logical window*/
char     cols;                  /*no. columns in logical window*/
char     yloc;                  /*initial row position*/
char     xloc;                  /*initial column position*/
unsigned reserved2;             /*system memory - overridden later*/
char     sharedprog [64];       /*name of shared program [ASCIIZ]*/
char     sharedata  [64];       /*name of shared data*/
char     controlbyte1;          /*control byte 1 -
                                  0x80 - writes directly to screen
                                  0x40 - foreground only
                                  0x20 - uses 8087
                                  0x10 - accesses keyboard buffer
                                  0x01 - swappable*/
char     controlbyte2;          /*control byte 2 -
                                  0x40 - uses command line params
                                  0x20 - swaps interrupt vectors*/

                                /*DESQview 1.00 extensions*/
char     startkeys [2];         /*starting keys from menu*/
char     scriptsize [2];        /*size of script file [bytes]*/
unsigned autopause;             /*pause after this many keyboard
                                  requests in one clock tick*/
char     disablecolormap;       /*1 -> disable color mapping*/
char     swappable;             /*1 -> application is swappable*/
char     reserved3 [3];
char     closeonexit;           /*1 -> close on exit*/
char     keyfloppy;             /*1 -> key floppy required*/

                                /*DESQview 2.00 extensions*/
char     DVPformat;             /*00 -> DV 1.2  and later,
                                  01 -> DV 2.00 and later,
                                  02 -> DV 2.20 and later*/
char     sharesmem;             /*1 -> uses shared system memory*/
char     physrows;              /*no. rows in initial physical window*/
char     physcols;              /*no. cols in initial physical window*/
unsigned expandedmem;           /*amount of avail EMS [kbytes]*/
char     controlbyte3;          /*control byte 3 -
                                   0x80 - automatically assign pos
                                   0x20 - honor max memory value
                                   0x10 - disallow close command
                                   0x08 - foreground only when graphics
```

```
                                    0x04 - don't virtualize*/
  char      kbdconflict;          /*keyboard conflict (normally 0)*/
  char      graphicspages;        /*no. of graphics pages*/
  unsigned  systemmem;            /*system memory size*/
  char      initmode;             /*initial video mode - default 0xff*/

                                  /*DESQview 2.20 extensions*/
  char      serialports;          /*serial port usage -
                                    -1 -> use all serial ports,
                                     0 -> uses no serial ports,
                                     1 -> uses port 1,
                                     2 -> uses port 2*/
  char      controlbyte4;         /*control byte 4 -
                                     0x80 - automatically close on exit,
                                     0x40 - swappable if not using ports,
                                     0x08 - virtualize text,
                                     0x04 - virtualize graphics,
                                     0x02 - share CPU in foreground,
                                     0x01 - share EGA when zoomed*/
  char      protectlevel;         /*degree of protection*/
  char      reserved4[19];
};
```

Reading a .DVP file in from disk is straightforward. .DVP files are most easily created using the Add a Program option under the DESQview Open menu option. All .DVP files are created in the \DV directory, but once there, they can be copied anywhere desired. Of course, it is also possible to read the default .DVP file in the above structure and then change some value, such as the initial directory, before executing the `app_start()` call.

The fact that the processes share neither code nor data space grants them a level of independence that is not possible with simple tasks. In the interest of saving memory, however, it is sometimes desirable that two processes share some code. This is possible using *shared programs.*

Shared programs are executable files specified in the .DVP file. Normally, a shared program is a loader used to make a DESQview oblivious program adaptable to a multitasking environment; however, shared programs have a unique property: before a shared program is loaded, DESQview checks to see if a copy is already resident in memory. If so, the already resident version is used again. The shared program stays in memory as long as at least one program that uses it is still executing. When the last program that loaded the shared program terminates, the shared image is unloaded from memory.

Although this is a long way from Dynamic Link Libraries (DLLs), a form of DLL can be implemented using shared libraries. For example, you might put common library routines into a shared program where they can be accessed from several processes, reducing the size of each process.

Memory Under DESQview

As we have seen, DOS loads itself as low as possible in memory. The first several hundred bytes are consumed by the interrupt table and BIOS data areas, then come DOS, its buffers, and any device drivers. Finally, any terminate-and-stay-resident (TSR) programs that the user has loaded appear. Any memory between the last TSR and the end of conventional memory is available for the application program's use under simple DOS.

When the user executes DESQview, DOS loads it into this memory area, as it would any other program. On a system with 640K or less of conventional memory, and without any expanded or extended memory, DESQview consumes roughly 150K (depending on configuration) of conventional memory; however, when either extended or EMS 4.0 memory is available, DESQview attempts to repay this memory penalty in several ways.

First, DESQview can increase the size of conventional DOS memory by noting that this area ends at 640K only because this is where the memory space reserved for video adapters begins. On machines equipped with either CGA or monochrome display adapters, actual video RAM does not begin until some 64 to 96K later. On PCs equipped with capable EMS 4.0 memory (or an 80386 or 80486 with QEMM386), DESQview can map memory into the region between 640K and the beginning of video memory to increase the space available to user programs. (Some EMS 4.0 cards do not support mapping memory into these lower ranges.)

In addition, DESQview comes with a special loader, XDX, which can load the vast majority of DESQview into memory above video RAM. This memory comes primarily from two sources. For PCs equipped with an 80286 and extended memory DESQview supplies a driver to tack the first 64K of extended memory (the HMA area) onto the end of the first megabyte. (This same function is handled by QEMM386 on 80386- and 80486-based machines.)

Even within the first megabyte there are several unclaimed areas. The blocks of address space claimed by the BIOS ROM, the hard disk ROM, and other plug-in cards, sit like islands surrounded by unused address regions. On PCs equipped with capable EMS 4.0 memory, the separately available QRAM driver can map memory into these areas. (Similar magic is performed by QEMM50/60 on the PS/2 models 50 and 60 and by QEMM386 on 80386- and 80486-based PCs.) XDX can reduce DESQview's overhead in conventional memory to less than 10K.

Since DOS programs must be in contiguous memory, DESQview cannot use any of the leftover upper memory for programs. To make this space available to the user, DESQview includes LOADHI.SYS and a LOADHI.COM. LOADHI.SYS allows the user to load small device drivers into this upper region, while LOADHI.COM does the same for TSRs. A separate FILES.COM and BUFFERS.COM allow the user to do the same for the DOS file and buffer spaces. In this way, the user can reduce the number of drivers loaded in conventional memory, leaving more for application programs.

These regions are represented graphically in Figure 7-5.

Once loaded into memory, DESQview consists of more than just DV.EXE. DESQview also builds a series of tables in an area called *common memory*. Common memory is located immediately above the low portion of DESQview, in the lower 640K. In addition to internal tables, DESQview stores objects and interprocess messages here. This memory area, like DESQview itself, is never remapped and so is available to all programs. The user may adjust the size of this area at setup time, but it is fixed once DESQview is loaded.

Every time a process is started, DESQview assigns it its own region of memory known as *process memory*. Process memory includes the program's code and data areas, the script buffers used to hold any scripts defined for the program, the context save area, where the CPU's registers are stored when the task is not executing, and the process' *system memory*. System memory is used to hold window buffers, panels, and messages between tasks in the same process.

One process may wish to share its system memory with another process. This is the case, for example, when one program wants to write into another program's window. To do this, you may include a "*" in the "Shared Program Pathname" field of the .DVP file menu. This causes DESQview to allocate system memory from an area called *shared system memory*. This area is kept immediately above common memory, and is available to all programs. (Any program that has a shared program also uses shared system memory.) Shared system memory starts out zero length and grows as processes require it.

Figure 7-5: The first megabyte of memory, showing areas where XDV might load DV.

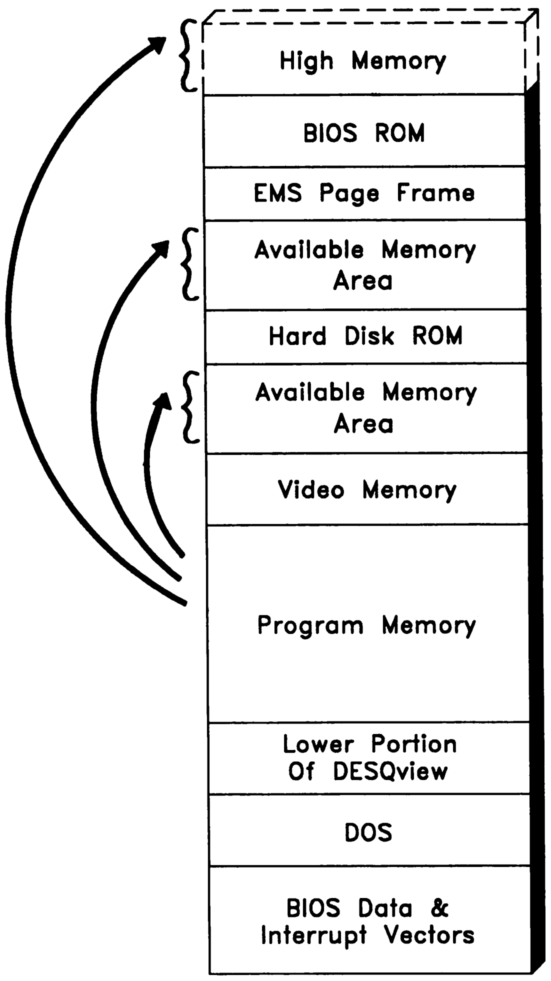
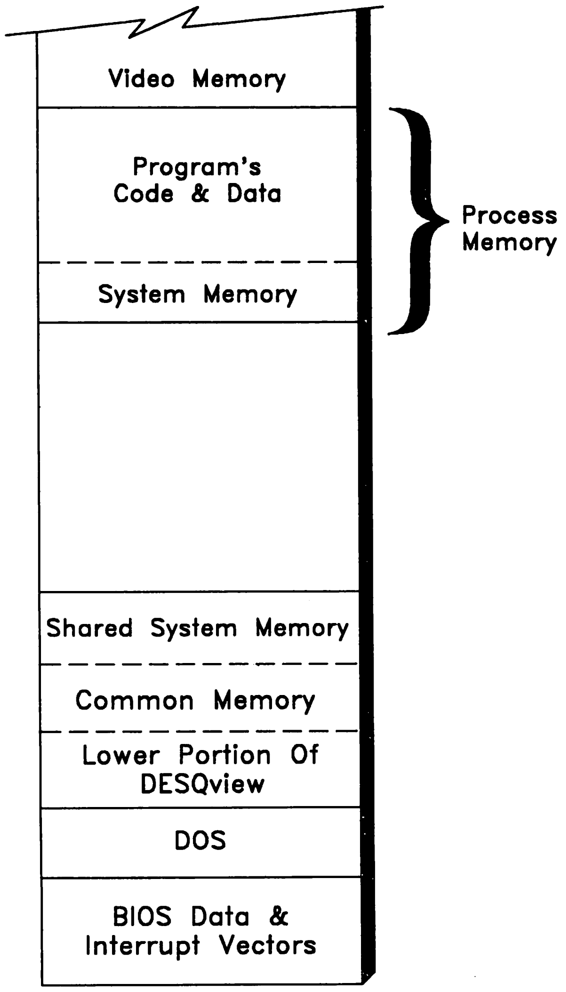


Figure 7-6 represents a single DOS application as it appears in memory.

When a program is executed, its .DVP file indicates how much memory it requires. DESQview could load the program low in memory and then set the end of memory value kept in the BIOS area to the end of the memory range dedicated to that program. This approach has two problems, however. First, few DOS programs ever check to see where this upper boundary is. If insufficient memory is available when they execute under DOS, they either simply do not load or else they crash. Second, most EMS 4.0 cards map the upper range of conventional memory, and not the lower range. Since DESQview wants as much of the application within EMS's mappable area, as we shall see, it is better to load the program as high as possible.

Figure 7-6: DESQview memory areas.



Therefore, when the user starts a program, DESQview allocates its process memory as high as it can in the lower 640K. The first program loads immediately below the video adapter memory at the end of conventional memory. The second application is generally loaded below the first, and so on. All of the programs loaded into conventional memory continue to execute.

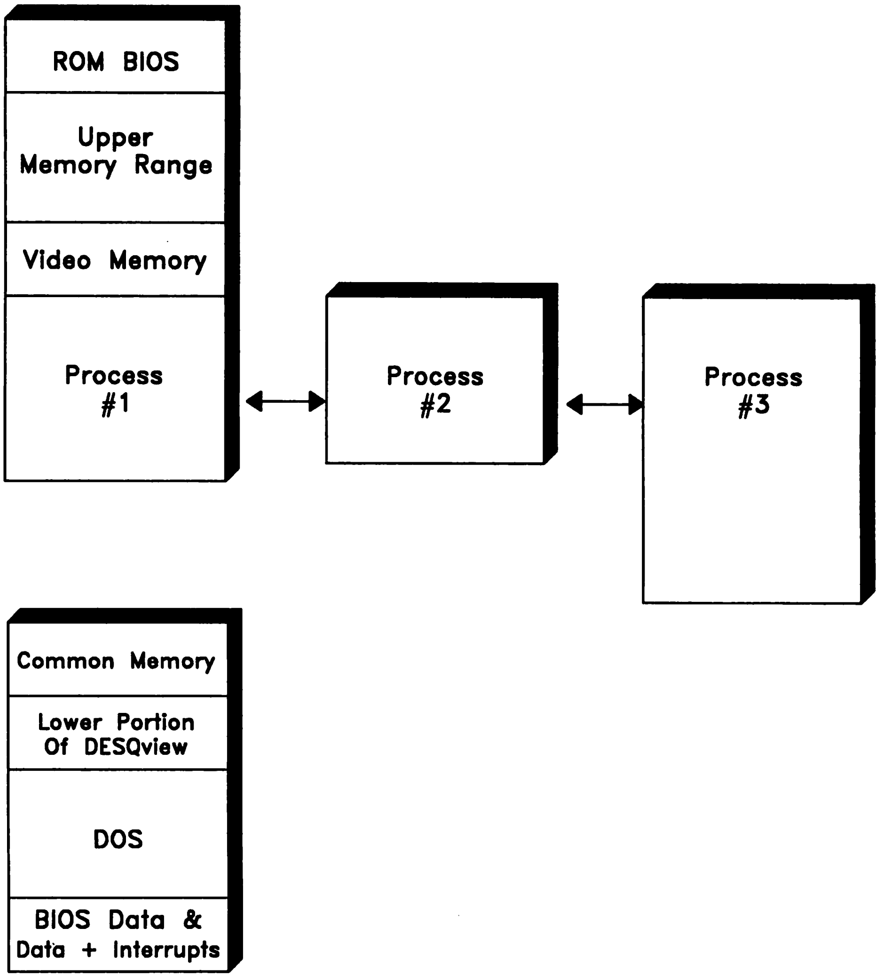
When the user attempts to load a program and DESQview determines that not enough conventional memory is left, DESQview attempts to free up memory by swapping out applications. Starting with the oldest application, and continuing with successively newer applications, DESQview continues to write the process memory of each application out until enough memory exists to load the new application. The user has some control over swap devices, but DESQview generally goes in order of decreasing speed. EMS 3.2 memory is the preferred swap device. If EMS 3.2 memory is not available, DESQview then begins swapping applications to a swap file on the disk. Applications that have been swapped out are "frozen" and do not continue to execute in the background; however, the user may switch them back into the foreground and, thus, back into conventional memory, at any time.

DESQview has much more flexibility with the loading of programs when EMS 4.0 memory can replace conventional memory in the lower 640K range. With today's machines, this generally involves disabling memory above some limit, via a switch on the motherboard.

DESQview uses the ability of EMS 4.0 boards to define multiple page frames of differing size and locations to map memory into any memory areas left unpopulated up to the beginning of video memory (this is the same process used to fill gaps in the memory space above video RAM). Rather than swap applications out to a swap device, DESQview can simply remap new memory pages into lower memory to make memory available for each new application. Since EMS memory can be remapped quickly, applications that have been mapped out of the lower 640K may be mapped back in to execute. Figure 7-7 represents this process graphically.

This slightly changes the approach DESQview takes in loading programs into memory. When it is time to load the first program, DESQview reserves as much EMS memory as indicated by the .DVP file and maps it immediately below video RAM. The program is then loaded and started in this area. When it is time to load the second program, DESQview repeats the process, again mapping the required amount of memory immediately below video RAM. Thus, each application occupies roughly the same logical address space in conventional memory. Of course, the first program must be mapped out of this space in order for the second to be mapped in. This makes it impossible for the two applications to access each other's memory directly. Expressed another way, a task in one process cannot access memory in a different process. Processes do share the lower memory areas, including DOS and DESQview (including command memory and any shared system memory).

Figure 7-7: Multitasking with EMS 4.0 memory.



Single applications larger than the EMS 4.0 page frame cause some problems, since they cannot be completely mapped out and must be swapped instead.

When an EMS 4.0 emulator is installed in an 80386- or 80486-based machine, *all* of available memory becomes swappable. This, in part, is why Quarterdeck bundles QEMM386 together with DESQview in the DESQview386 package. With

DESQview386, it is not necessary to disable motherboard memory. In addition, since there is no limit to the size of the EMS page frame, it is not possible for an application to be larger than the page frame and, therefore, unmappable. (An application may be too large to fit into the remaining 640K at all, of course.)

With DESQview386, the user may load as many applications as can be held in available memory. A 2-megabyte machine can execute four or five normal sized applications simultaneously, before memory is exhausted. Once available memory is depleted, applications are swapped to disk as needed.

DESQview386 can use some of the protection features of the 80386 processor as well. For example, if the user desires, DESQview386 can trap any attempt by an application to write memory outside of its own process. Except for these protection features, DESQview386 treats 80386 memory as EMS 4.0 memory.

Intertask Communication

When we started as programmers, especially if we started with BASIC, our programs tended to be one large monolithic whole with little or no internal structure. Eventually we came to see the advantage in dividing our programs up into functions. This allowed us to break a problem down into a series of smaller problems, each of which could be dealt with separately, with a resulting savings in time. As soon as we broke our programs into functions, however, the problem arose of how these functions should communicate.

The first solution was simply to use the same techniques for communicating that had worked between blocks of code before. One function simply leaves some value in a global variable so that the other function can conveniently find it there when it is called. Soon we discovered that this was not the ideal means of communication. While global variables are fast, it is generally far better to forgo the speed for the increased control and safety of passing data to and from functions via arguments. This is the primary means of communication between functions in a single-tasked environment.

In some multitasking environments, it is possible to pass arguments to tasks in exactly the same way as they are passed to simple functions. Unfortunately, this is not the case in DESQview. The `tsk_new()` API call accepts the address of a function that takes no arguments and returns nothing (the fact that in the prototype this function is of type `int` is a throwback to K&R—in fact, the function should have been declared to be of type `void`).

In the absence of function arguments, the question arises of how we should communicate between tasks within a single process. Our first response might be to drop back to past experience and simply use global variables again. The parent task can store a value into a global variable before starting the subtask. Once the subtask is started, it examines the location to find the data. This works for data that is set once and does not change thereafter, but what about changing data?

If more than a single value is to be communicated, the two tasks must agree on some sort of protocol. They may agree on what constitutes "no data," which can then be stored into the location once data has been read out, as an indication that it is now available to hold new data. If this is not possible, a second location may be used as an access flag. For example, a flag value of zero might correspond to "no data," while a nonzero flag indicates data is waiting to be accessed.

If the communication is two-way, in that both tasks can write into the location, data may get overwritten and lost in the following way: the parent task reads the location and checks that it is currently set to "no data." Before it can write its data into the location, however, its time slice comes to an end. When the other task starts its time slice, it reads the same location and also sees that it contains the "no data" value. This second task then writes its data. Now when the first task gets control back, it continues with the write operation that was suspended before, overwriting the second task's data. (This is very similar to the `customer_balance` problem discussed earlier.)

The loss of data due to its being overwritten by another task is known as a *data collision*. In some applications, the occasional loss of a data word is insignificant, but for the majority of programs this is a serious problem. For these programs, a section of code where such a data collision might occur is known as a *critical region*. The specific operation—in this example the reading and writing of a variable in memory—is called the *critical operation*.

There are several ways to avoid data collisions. Noticing that the problem only arises when two or more tasks write to the same location offers one solution. Often, we can split communications into two separate paths. In this case, the parent task might use the location `datato` to send information to the subtask while the subtask might use the next location, `datafrom`, to send information in the opposite direction. If each task reads both locations but only writes to one of them, the collision is avoided.

Often it is not possible to agree on a protocol or divide the global data up in such a way as to guarantee that a collision will not occur. This is usually the case when more than two tasks are potentially involved, or when critical regions arise around objects other than memory locations.

Suppose, for instance, that one application decides it needs to send output to the printer. While it is printing, another task detects an alarm condition and decides to print the warning message. If access to the printer is not controlled, what appears on the printer will be a mix of the output of the two tasks, and the alarm warning may be illegible. Thus, the print operation represents a critical region between these two tasks. These types of critical regions cannot be protected with "no data" type protocols, since they do not represent memory locations and cannot be read.

It is clear from our examples that collisions occur because the one task is so unfortunate as to get rescheduled during a critical operation. The most obvious solution then is for a task to disable rescheduling before beginning a critical operation, and only reenable it after it has completed the operation.

In DESQview, the `api_beginc()` API call ("beginc" stands for "begin critical operation") disables rescheduling. Once rescheduling has been disabled, a task is assured that it will retain control until it either reenables scheduling via an `api_endc()` API call or performs a system call that suspends the task. (If DESQview left scheduling disabled when the task suspended itself, the system would hang forever, since no one would be allowed to run.)

Wrapping a critical region in `api_beginc()` and `api_endc()` calls is a common ploy. It appears to be fairly straightforward and so is very popular with programmers new to multitasking. What they quickly realize, however, is that it is often a very unsatisfactory solution. One problem is that leaving rescheduling off for long periods adversely affects the performance of the system—imagine, for instance, if rescheduling were disabled for the entire time it took to print a letter, just so another print operation could not start. If the critical operation takes very long to perform, the user notices the system halt and lurch as rescheduling is turned on and off.

A worse problem with this approach is that it is voluntary. A task might come along and access the controlled memory location or device without disabling rescheduling first. Such a violation is not easily detected. Suppose, for example, that the print warning message module in the above example printed without disabling rescheduling. Module testing is not likely to reveal the problem. It is not until another task decides to begin printing just after an alarm starts printing that the user begins complaining of lost warning messages on the system. Rare collisions around data variables resulting in incorrect bank balances or a crashing system are even worse and even more difficult to find.

A task may violate this protocol unintentionally as well. A function called while scheduling is disabled may inadvertently reenable it by performing a seemingly innocuous system call—any DOS call can result in the task being suspended and scheduling reenabled, for instance. While the programmer may have the best of intentions, it is difficult to make absolutely sure that rescheduling remains disabled over a large segment of code.

Still, reading data out of a global variable or flag, checking it, setting it, and storing right back does not take very long, and it should not require very many instructions. Therefore, protecting access to such common memory locations by disabling rescheduling temporarily is an acceptable practice.

Larger critical areas may be protected by a flag, often called a *lock*. The idea is to assign a flag to each critical operation. Before entering the critical section of code, the program checks the flag. If the flag is clear, it is safe to proceed into the critical section. The task then sets the lock to ward off other tasks that may attempt to enter and continue on. Once the task is finished, it clears the flag and proceeds. Obviously, the checking and setting of the flag is itself a critical operation and must be protected with an `api_beginc()` call. In DESQview, such an operation is as follows:

```
int latchvar;
void latch (void)
{
     for (;;) {
         api_beginc ();           /*disable scheduler*/
         if (latchvar == 0)       /*if latch clear...*/
             break;               /*...continue; else...*/
         api_endc ();             /*...reenable scheduler...*/
         api_pause ();            /*...and give up control*/
     }
     latchvar = 1;                /*set latch and return*/
     api_endc ();
}

void unlatch (void)
{
     latchvar = 0;
}
```

This approach to collision avoidance is much better than the simple disable rescheduling approach. Here, tasks not attempting to enter the critical region are not affected. While a task that is attempting to print might be suspended for a long time waiting for another to finish, all of the tasks that are not trying to print are not affected. The system continues to execute smoothly. However, a new problem arises with this approach. Requesters are not granted access in the order in which they request it.

Suppose, for example, that three tasks all attempt access to the printer. Task A requests the latch and is granted it. Task B and then task C request the latch but must wait. When task A completes printing and clears the latch, there is no guarantee that task B will get control next, even though its request preceded that of task C. The order in which tasks B and C are granted access is completely random. In fact, if task A restored the latch but then decided to print again before its time slice was completed, it would be granted clearance by the cleared flag before either task B or task C could "wake up" and grab it.

An even worse problem is common to all of these solutions. They all rely upon the fact that the several tasks involved all have access to some global variable. With tasks that share the same process memory this is true, but for tasks in different processes, this cannot be the case. Processes do not share code or data segments. For these types of tasks, DESQview offers a different communications path: intertask messages.

Intertask Messages

The principle behind intertask messages is simple. The sending task simply bundles up the information to be transmitted in a locally defined structure and mails it to the other task using a `mal_write()` DESQview API call. The message is copied into a mailbox maintained for the other task in system memory (messages between tasks in separate processes are saved in common memory). These mailboxes can hold a number of messages in a first-in-first-out queue. The receiving task may read these messages out of its mailbox at any time by executing a `mal_read()` API call. Each subsequent `mal_read()` call returns the next message. If the mailbox queue is empty, `mal_read()` suspends the caller until a message appears. If a task does not wish to be suspended, it may poll the mailbox first, using the `mal_sizeof()` call. This call returns the number of messages in the queue. If the count is nonzero, the task can `mal_read()` the mailbox without being suspended.

Intertask messages avoid most of the intertask communication problems inherent in other communications paths. First, all access to common locations is controlled by the operating system, making data collisions impossible. When a task sends a message, it is copied out of its space. Like a variable passed to a function by value, any subsequent changes it makes to the message buffer are not copied to the sender. Second, tasks do not have to wait to send messages. A task may send a message and continue, even if other messages are already queued up for the task to work off. Third, the order of requests is retained. If task A sends a message before task B, then task A's message gets processed first, despite the randomness of task scheduling.

Finally, intertask messages, like function calls, represent a "point to point" communication path. That is, when task A sends a message to task B, it is clear where the message is going and who will receive it. It is not possible that some task C might come along and intentionally or unintentionally interfere with this communication path. Not only is this important to the programmer writing the program, it is equally important to the person who must come along years later and maintain the program. This can make a very large difference in the ability to maintain large programs.

When a task is started, along with the default window and keyboard objects that DESQview opens, it also opens a default mailbox object. The handle of the default mailbox associated with any task can be returned from the `mal_of()` API call. A task may know its own mailbox handle by executing the `mal_me()` call.

In our CLOCK program, the function `program_body()` starts the subtask `ticktock()` to update the time in the Clock window. In order for `ticktock()` to write to this window, however, it must have the window's handle. `Program_body()` could have simply stored the Clock window handle in a global variable, which `ticktock()` could then have accessed. Instead, `program_body()` stores the handle into the structure "message" and sends it to `ticktock()`'s mailbox as in the following code segment:

```
ulong panelwin;                 /*window handle for the clock*/
 ulong malhan;                  /*mailbox handle of subtask*/
 struct windowmsg message;      /*message to send to subtask*/

 tskhan = tsk_new (ticktock, taskstack, STACKSIZE,
                   "", 0, 0, 0);
 malhan = mal_of (tskhan);
 message.windowhandle = panelwin;
 mal_write (malhan, (char *)&message, sizeof message);
```

For its part, `ticktock()` reads the message immediately upon starting, as in the following:

```
ulong winhandle;
struct windowmsg *winmessage;
int              messagelength;

/*read the message with the window handle*/
mal_read (mal_me(), &(char *)winmessage, &messagelength);
winhandle = winmessage -> windowhandle;
```

Notice that the message is copied out of the sender's memory area and into system memory when it is sent. The caller is free to reuse that space as soon as control is returned from the `mal_write()` call. It is not copied into the receiver's memory area, however. Instead the receiver is given a pointer to the message in DESQview's memory. This is why the second argument to `mal_read()` is not the address of a message buffer but "a pointer to a pointer" to a message (the cast `(char *)` is only so the call matches the prototype declaration). The data in the message stays valid until the next time the task performs a mailbox read. Therefore, `ticktock()` copies the data out of the message and into local, safe storage.

(Incidentally, CLOCK also uses a global variable communication path when `ticktock()` stores the current time in the `currenttime` structure, which is subsequently used within the parent task. It is valid here since the parent task never writes this structure, thus precluding any chance of collision.)

This point-to-point message scheme means that both the sender and receiver must know the mailbox's handle. A task can always find its own handle, using `mal_me()`. A parent task can always find the default mailbox of any of its subtasks by using the `mal_of()` API call on the task's handle.

However, a task may open up additional mailboxes using a `mal_new()` call, and these are not associated with the task handle. To solve this problem, a task may assign any mailbox a name using `mal_name()`. Any other task can then find the handle of this mailbox by looking up its name via the `mal_find()` call. This can considerably enhance the readability of the resulting code considerably. For example, `ticktock()` could name its mailbox "Update Clock Display." Then any task that wanted its display updated on a regular basis could send a message to the Update Clock Display mailbox.

This is very common in applications fraught with collision problems, such as a database. If every task in the program can access the database, controlling all the locks necessary to avoid database contention can become a real problem. It is easier to name one task as the database engine and allow it to perform all reads and writes, acting as a form of traffic cop to avoid data collisions. This database engine may even be in a separate process—in fact, this process may not even be on the same machine! Assigning the mailbox for this task a name like "Database" allows each of the other tasks to find its handle in order to make requests. The intent is clear to even the most casual reader what a task sending requests to a mailbox named Database is trying to do.

Of course, the database must be able to answer the requests. The requesting task can simply store the mailbox handle where it expects its answer in the request message, but this is not necessary. If a task receives a message, calling the `mal_addr()` API function returns the mailbox handle of the task that sent the message, so it is possible to reply to messages even if you do not otherwise know from where they came.

The overhead of copying large messages into system (or common) memory during the `mal_write()` may become objectionable. DESQview does allow messages to be passed by reference with the `mal_subfrom()` call. That is, when the receiving task receives a message address, it is, in fact, pointing directly at the message in the calling task.

This can be very dangerous, however. First, the calling task may not reuse the message space until the receiving task is through with it. For this reason, some protocol must be established for returning the buffer to the originating task.

Second, the receiving task must have access to this memory. This is no problem for tasks within the same process; however, for tasks in other processes, this is a real problem. Such a task would receive a logical address to a region that may not be mapped into memory when the task is. Since how the processes are mapped is a function of the hardware type and configuration, this is the type of problem that might not appear during development, only showing up on user's equipment.

This problem can be avoided by using shared system memory, that is, specifying a "*" in the .DVP for each process and then allocating memory from this area in which to build the message via the `api_getmem()` API call. Since shared system memory comes out of the lower 640K, however, there is not enough room for truly large messages. In systems with EMS memory, it is just as simple to allocate a block of expanded storage in which to build the message. The handle for this EMS storage can then be inserted into a normal message and sent to the other process. This process then maps the EMS memory into its page frame and accesses the message directly. This allows messages to be built up to the size of available EMS memory.

Notice that either way, the sending task may continue to access the memory area, even as the receiving task is using it, as long as a protocol is established to avoid collisions. In fact, this is exactly the same collision problem as that presented by global variables between tasks. Passing EMS handles or pointers to shared system memory allow processes to establish the same common memory communication path available to tasks, but brings with it identical problems.

Avoiding access by multiple tasks to global variables only eliminates the data collisions. The intertask messages we have seen so far cannot solve non-data collisions such as those that might arise around devices. We have already seen how we can use a lock variable to control access between tasks within the same process. The same method could be applied to tasks in different processes if we first allocate a small section of shared system memory and pass its address to all of the tasks requiring access via messages. Mailboxes provide a better solution, however.

In some message-based operating systems, control to such a critical region can be controlled in the following way. The parent task first creates a "lock mailbox" and sends it a single dummy message known as a *semaphore*. Any task that desires entry into the critical region must first read the semaphore from the mailbox. Once the task has read the message, it may continue. Once it has left the critical section, it must then send a dummy message back to the mailbox to restore the semaphore. If a second task reads the mailbox while the first task is in the critical region, the semaphore message is not there (having been read by the first task) and the second task is suspended. When the first task returns the semaphore, the second task is unsuspended and allowed to proceed.

Similar in concept to simple locks, semaphore mailboxes have the same advantages that normal intertask messages have over global variables: tasks are not rescheduled needlessly, requests are processed in order, tasks need not be in the same process, and so on. Under DESQview, however, two tasks may not actually read the same mailbox. Therefore, DESQview defines two special API calls to allow mailbox semaphores to be established: `mal_lock()` and `mal_unlock()`. The first task to perform a `mal_lock()` API call on a mailbox is allowed to pass. Subsequent tasks that attempt to `mal_lock()` the mailbox are suspended until the first task performs a `mal_unlock()`. (Actually, the first task may `mal_lock()` a mailbox multiple times; the lock is not cleared until the same number of `mal_unlock()`s have been performed.) The same mailbox may not be used as a normal message mailbox and a semaphore mailbox; once a task has locked a mailbox, it may no longer read it or send it a message.

How DESQview Uses Messages

In our discussion of messages it may have struck you that reading a mailbox object is a lot like reading the keyboard under DOS using BIOS calls. Performing a BIOS read command of the keyboard suspends the calling program until a key arrives. A program can simply poll the keyboard to see if a key is present to avoid being suspended. And finally, keystrokes arriving from the keyboard are queued up into the keyboard input buffer. This is not very surprising since the keyboard queue is designed to handle exactly the same problem as message mailboxes: communication between two independent entities. In this case, however, the two entities are the single DOS task and the user.

DESQview encounters the same problem as DOS with almost all of its Input/Output paths. It is not very surprising, then, that DESQview uses intertask messages for the majority of its communication with application software. A task that performs a `key_read()` of the keyboard object is merely reading a message from keyboard mailbox. Every time the user strikes a key, DESQview places the key into a message and sends it to a keyboard object.

DESQview generally uses the position of a task's windows to decide which keyboard object gets the message. DESQview sends keyboard messages to the task that has the window nearest the foreground. Usually this is the foreground window (the window marked with a double border). Thus, the foreground task "hears" the keyboard and the others do not.

Just as DESQview can send keystroke messages to a keyboard object, an application task can `mal_write()` a message to another task's keyboard object just as it might to the task's mailbox object. The mechanism is exactly the same. The effect on the receiving task is as if the user had typed in whatever the sending task sent.

In fact, DESQview itself sometimes sends messages other than simple keystrokes to a keyboard object. The reader may have noticed that the `pan_apply()` API call used to display a panel in CLOCK returned not only the window handle used to manipulate the panel's window but also the handle of a keyboard object. This keyboard handle allows a task to input from a panel as well as output to it. DESQview handles the actual user input, the manipulation of the mouse, cursor keys, and so on. There are several modes of operation for panels, but in the most common mode, the application task receives a message containing simply the field number followed by what the user entered in that field.

Panel Messages

There are three types of fields under DESQview: output fields, input fields, and select fields. The user may not enter data into output fields. The user may enter free text into an input field. When the user presses the Enter key, DESQview sends a message containing the field number along with the contents of the entire field, and moves the cursor to the next input field. Special flags allow the message to be sent as soon as the user exits the field (even without pressing Enter), to automatically convert everything to uppercase, to right justify input, and so on. DESQview does not support a flag to allow only numerical input to a field.

The users may select a select field by clicking on it with the mouse, by placing the cursor on the field and pressing Enter, or by entering the special "select" character defined for the field. No other form of input is possible. To inform the user when a select field is properly pointed at, DESQview changes its color to the "pointed at" color. When the field is selected, it changes again to the "selected" color. Reselecting an already selected field, causes it to deselect, reverting back to the "normal" color. When a select field is selected, DESQview sends a message containing the field number and a "Y"; when the field is deselected, DESQview sends the same message with an "N".

Referring back to our CLOCK program, the `definealarm()` function, which reads the alarm clock input, appears as follows:

```
/*definealarm - put up the alarm set panel where the user
               may enter the alarm time*/
void definealarm (ulong panhan)
{
     ulong alarmwin, alarmkey;
     struct panelmsg *alarminput;
     int             inputlength;

     /*first, open the ALARM panel to display Alarm Set window*/
     if (!pan_apply (panhan, win_me(), "ALARM", 5,
                         &alarmwin, &alarmkey)) {

          /*update the window to the previous alarm time...*/
          updatetime (alarmwin, &alarmtime);

          /*...and position the cursor for time entry*/
          fld_cursor (alarmwin, FLD_HOURS);

          /*now wait for the user to update the time fields*/
          for (;;) {
               key_read (alarmkey, &(char *)alarminput, &inputlength);
```

```
                    switch (alarminput -> fldid) {

                   /*selecting fields 2 thru 5 just
                     fills values into the alarm time*/
                    case FLD_HOURS:
                        strncpy (alarmtime.hour, &alarminput -> data, 2);
                        break;
                    case FLD_MINUTES:
                        strncpy (alarmtime.min,  &alarminput -> data, 2);
                        break;
                    case FLD_SECONDS:
                        strncpy (alarmtime.sec,  &alarminput -> data, 2);
                        break;
                    case FLD_AMPM:
                        alarmtime.afternoon = alarminput -> data;
                        break;

                    /*selecting field 1 removes the alarm window and
                      returns control to the main program*/
                    case FLD_ALARM:
                        win_free (alarmwin);
                        return;
                    }
               }
          }
     }
```

Applying the Set Alarm panel returns the keyboard object handle `alarmkey`. The next two calls simply update the time display in the Set Alarm window to the previously entered alarm time and position the cursor at the beginning of the time field to simplify entering the time. The function then begins reading messages from the panel. When the user enters data into any of the input fields, DESQview generates a message containing the one or two ASCII characters entered. These are dutifully copied into the structure `alarmtime`. `Definealarm()` responds to a message from field 1, the asterisk, by removing the panel from display with `win_free()` and returning to the caller.

Input fields are a powerful part of panels. The application program is spared the difficulty of writing code to handle arrow keys, mouse movement, deletes, overstrikes and the rest. The resulting program is smaller and easier to write, but there is a further advantage. All program written using panels for input have the same "look and feel." In a given situation, pressing the Tab key, for example, has a predictable result, no matter what the application. If you wonder what that look and feel is, try popping up the DESQview menu—all of the DESQview windows are panels.

Notification Messages

DESQview can sometimes send messages to a task's mailbox object as well. When a task displays a window on the screen, normally, the user is allowed to move it, resize it, and scroll it at will. For some applications, this is unacceptable. In this case, the task may disable these functions for the particular window. In other cases, it is enough that DESQview inform the application task that its windows are being manipulated. This is called turning on *notification* for a particular window and is handled via the `win_notify()` call. The notification flags are defined in Table 7-1 below.

Table 7-1: Notification types available for DESQview windows.

Type	Notify On
NTF_HMOVE	The window is moved horizontally
NTF_VMOVE	The window is moved vertically
NTF_HSIZE	The window is resized horizontally
NTF_VSIZE	The window is resized vertically
NTF_HSCROLL	The window is scrolled horizontally
NTF_VSCROLL	The window is scrolled vertically
NTF_CLOSE *	The window is closed
NTF_HIDE *	The window is hidden
NTF_HELP	Help is selected from the DESQview menu
NTF_POINTER	A message is sent to an application
NTF_FORE *	The window is brought to the foreground
NTF_BACK *	The window is brought to the background
NTF_VIDEO *	Video adapter changes mode
NTF_CUT	Cut is requested
NTF_PASTE	Paste is requested
NTF_DVKEY *	DESQview key is entered
NTF_DVDONE *	DESQview menu is closed

* indicates notification sent no matter where mouse points

Once a notification event is enabled with `win_notify()`, DESQview sends a message to the default mailbox object of the task that opened the window if: a) the notification event occurs, and b) the mouse pointer is pointing to the window. (For those notification events marked with an asterisk, the second requirement is not necessary.)

This is sometimes a very useful capability. For example, a DESQview-specific word processor might want to be informed if the user attempts to close it from the DESQview menu. When the close message arrives, the editor can then ask the user whether it should save the file being edited before the application is terminated and the edits lost. (In addition, the word processor might disable notification of the Close Window command when the file has not been changed, such as immediately after a save. Closing the program at this point will not lose data. The program might then reenable notification of the Close command when the buffer has been changed and data could be lost.)

An even more interesting notification event is the Help event. With notification of this event enabled, the application receives a message whenever the user enters the Help command on the DESQview menu. This knowledge lets a user application add its own Help capability to that of DESQview.

Messages to Other Object Types

Besides using the keyboard and mailbox objects, DESQview uses two other object types to communicate with the user application: the pointer object and the timer object.

The pointer object is the interface between the program and the "pointing device." The location of the mouse pointer is indicated by a white diamond on the screen. The pointing device might be a mouse, but DESQview also supports something known as a "keyboard mouse." When using the keyboard mouse, pressing the "mouse key" causes the diamond to appear on the screen. It can then be moved about with the arrow keys. Pressing the mouse key once more returns the arrow keys to their normal use.

After creating a pointer object using the `ptr_new()` API function, an application must associate that pointer with a window, using the `ptr_open()` call. Doing so informs DESQview that pointer messages should be sent to that object whenever the pointer is within that window. DESQview normally generates a message whenever the mouse moves enough to cause the mouse pointer to change rows or columns. An application cannot determine mouse resolution finer than a row or a column. The message received contains the row and column of the mouse, plus the status of both the left and right mouse buttons.

The `ptr_addto()` API call can set flags associated with the pointer object. For example, one flag tells DESQview that the application is not interested in mouse movement, but only in mouse clicks. In this case, DESQview only sends a message when the user clicks either or both mouse buttons. Other flags control whether a message is sent both when the mouse buttons are pressed and released or simply when they are pressed, whether the screen location should be screen relative or window relative, and whether the mouse diamond should be visible or invisible. These flags are outlined in Table 7-2.

Table 7-2: Pointer flags.

Flags	Description
PTF_CLICKS	Only report pointer clicks (default is to report mouse movement as well)
PTF_UPDOWN	Generate message both on press and release(default is press only)
PTF_RELSCR	Row and column are screen-relative (default is window-relative)
PTF_MULTI	Multiclick messages (accumulate rapid clicks and report them together instead of individually)
PTF_BACK	Generate messages even when app is in background
PTF_NOTTOP	Generate messages even when window is not top-most in application
PTF_HIDE	Hide pointer diamond when in window

Normally only the top window of the top-most application receives pointer messages. This keeps the user from selecting a menu option when a submenu has been opened on top of it. An application can specify, however, that a task wants to receive pointer messages even if it is not the top-most window. In fact, a window can indicate that it wants to receive pointer messages even if its application is not top-most; that is, it is executing in the background. Of course, no matter what is selected, the application only receives messages when the pointer is within a visible portion of its associated window.

Notice that although our CLOCK program accepts mouse input, since it performs all input through the panel, it can allow DESQview to handle the mouse. CLOCK, therefore, does not open a pointer object itself.

The final object type is the timer object. As the name implies, user applications use this object type to measure or delay specific lengths of time. A timer object is first created with a `tim_new()` API call. It may then be set to go off either for a specific time of day, using the `tim_write()` function, or after a given duration, using the `tim_addto()` call. Once a timer object is set, a task may then read the object using `tim_read()`.

Just as with any other object, reading an empty timer object results in the calling task being suspended. When the timer expires, DESQview sends a message to the object, and the task is unsuspended. The task does not continue execution until the next time it is scheduled. A task may "peek" into a timer object to see how much time is left with a separate API call.

CLOCK uses both forms of timer object. The `ticktock()` function delays for one second at a time between updates of the clock window, while `setalarm()` sets a timer object to expire at the user-selected alarm time. A portion of `ticktock()` appears below:

```
ulong timpause;

timpause = tim_new();
for (;;) {
    /*start a 1 second timer and wait for it to expire*/
    tim_addto (timpause, 100);
    tim_read (timpause);
          .
          .
          .
}
```

With so many different types of objects about, servicing them all could get quite confusing. Consider, for example, a program that poses a question to the user and awaits input. Suppose that, just to be user friendly, this program puts up a help screen if the user selects Help or if there is no input for 10 seconds. To do this, the task must read three different objects: the keyboard object from where input is to come; a timer object, which will go off after 10 seconds; and the default message object, where DESQview will send a message if the user selects Help.

The task cannot simply read any of these objects, since doing so suspends the task until a message appears in the object read, irrespective of what might happen to the others. The task could poll each of the objects using the `_sizeof()` calls in a loop, rescheduling after each pass via an `api_pause()` call to allow other tasks a chance to execute. As soon as `_sizeof()` returns a nonzero count for any of the objects, the task can then read it out without fear of being suspended.

This isn't a very clean solution, and it wastes CPU time. The task must repeatedly be scheduled to run even though all that is likely to happen is that it will check three empty queues and then give up control again. It would be better if the task could suspend itself until a message appeared in *any* of its objects.

For this, DESQview defines a special object known as the *object queue*. The object queue is simply a queue of object handles. Unlike other object types, a task has only one object queue. (It is created automatically when the task is started, and may not be deleted with an `obq_free()`, nor can new ones be added with an `obq_new()` call.) Every time a task creates a new object (or has one created for it;

for example, when it applies a panel), the object is added to the object queue's list. Whenever a message is sent to any of these objects, the handle of that object is placed in the object queue.

Reading the object queue via an `obq_read()` suspends the calling task. When a message appears in any of the task's objects, the handle of the object is returned from the `obq_read()` call. This handle is then compared with the handles returned from the previous `_new()` calls to determine which object has received the message. Objects may be prioritized under DESQview 2.20 and later, so that certain objects go to the front of the list to be serviced even when other handles are already queued up.

CLOCK reads the object queue when waiting for either keyboard input from the Clock panel or for the alarm timer when the alarm clock expires. The code to do that appears below:

```
for (;;) {
     /*now wait for an event to occur*/
     obqhan = obq_read();
     if (obqhan == panelkey)    /*keyboard input*/
         alarmonoff (panhan, panelwin,
                     timalarm, panelkey);

     if (obqhan == timalarm)    /*alarm timer*/
         declarealarm (panelwin);
}
```

This is the normal structure of a DESQview specific task with multiple inputs.

Why use DESQview?

As a DOS-based multitasking environment, DESQview has its limitations. It does allow the user to run DOS extender programs that are compatible with the Virtual Control Program Interface (VCPI), but otherwise provides no support for single programs larger than 640K comparable to the memory management in Microsoft Windows. It does not run in protected mode on 286 CPUs, so it cannot take advantage of extended memory on 286-based machines. And its character mode-based windows and panels are not as attractive as the graphics-mode displays of Windows.

But weighed against these drawbacks, DESQview provides an elegant object-oriented model of communication, both between application tasks and between applications and the multitasking environment. The DESQview API is straightforward and easy to learn. The DESQview Panel Design Tool allows rapid development of uniform but highly adaptable windows for information display and data entry. Most importantly, DESQView's overhead is low and its hardware requirements are small. Applications run under DESQview retain 90 percent of their performance on plain non-multitasking DOS, DESQview occupies a scant 200K of RAM (including DOS), and can (unlike Windows) be used effectively on 8086/88 machines.

In short, DESQview is an environment capable of ushering programmers into the next generation of DOS-based application software, without giving up support for the last generation of software and the huge existing base of 8086/88-based PCs.

Chapter 8

VCPI for EMS/DOS Extender Compatibility

Robert Moote

VCPI (Virtual Control Program Interface) is a program interface that allows EMS emulators and DOS extenders to coexist on an 80386- or 80486-based PC. The interface is defined as an extension to EMS 4.0; it consists of a set of calls provided by the EMS emulator and used by the DOS extender. Without the cooperation made possible by the VCPI interface, a user could not run an extended DOS application on a machine with an EMS emulator installed.

From the typical PC user's point of view, there is little need to understand or even know about VCPI; the whole purpose of the interface is to allow programs to cooperate without the knowledge of, or intervention by, the user. Software developers creating protected-mode applications using a DOS extender, on the other hand, should at least be aware of the existence of VCPI, because it means they need to test their products in two environments (DOS, and DOS with an EMS emulator providing VCPI). An understanding of VCPI is important to software developers who wish to provide or use the VCPI interface directly in their products; this applies chiefly to developers of EMS emulators and DOS extenders, since the interface is tailored specifically for those two classes of program. For any programmer, an examination of the VCPI interface can yield some interesting insights into the compatibility problems encountered by protected-mode software running under DOS.

VCPI support is included in most popular DOS extender and EMS emulator products. Tables 8-1a through 8-1c list products that provide VCPI support. Table 8-1d lists products for which VCPI support has been announced in new releases due out in the first quarter of 1990. These products should be available by the time this book is published.

There are two primary sources of conflict between EMS emulators and DOS extenders. Both types of program switch the processor to protected mode, and both typically allocate much or all of available extended memory (memory above 1 megabyte). Most of the VCPI interface is devoted to providing a means for co-operation in these two areas, and most of the discussion here is directly concerned with one or both of these problems.

In this chapter, we will look first at the contention that arises between EMS emulators and DOS extenders for extended memory and protected mode. Next, we will discuss the interface itself and examine an example of a DOS extender that uses the interface to run under an EMS emulator. Finally, we will scrutinize the internals of the interface to see what makes VCPI work.

Table 8-1a: EMS emulators supporting VCPI.

Product Name	Company
386-to-the-Max	Qualitas, Inc.
CEMM	Compaq Computer Corporation
HPEMM 386	Hewlett Packard
HPEMM 486	Hewlett Packard
QEMM-386	Quarterdeck Office Systems

Table 8-1b: General-purpose DOS extenders supporting VCPI.

Product Name	Type	Company
386 I DOS-Extender	32-bit	Phar Lap Software, Inc.
DBOS/386	32-bit	Salford Software Marketing LTD., for use only with Fortran programs compiled with FTN77/386 from the same company.
DOS/16M	16-bit	Rational Systems, Inc.
OS/286	16-bit	Eclipse Computer Solutions, Inc.
OS/386	32-bit	Eclipse Computer Solutions, Inc.

Table 8-1c: Proprietary DOS extender technology supporting VCPI.

Product Name	Company
Professional ORACLE	Oracle Corporation

Table 8-1d: Q1 1990 EMS emulators with VCPI support.

Product Name	Company
ALL CHARGE 386	All Computers, Inc.
HI386	RYBS Electronics, Inc.
Turbo EMS	Merrill and Bryan

Incompatibilities Between EMS Emulators and DOS Extenders

Both EMS emulators and DOS extenders are available for 286-, 386-, and 486-based PCs. To see how conflicts arise, let's review how these products work.

DOS extenders allow large applications to run under DOS by making it possible for them to run in protected mode. A DOS extender provides a protected-mode environment on top of the standard DOS environment, and also directly allocates extended memory for use by the application program. While implementation details differ, this process is fundamentally the same on a 286 machine as on a 386 or 486.

EMS emulators are software-only products that turn extended memory into expanded memory. An EMS emulator directly allocates a chunk of extended memory and makes it look like expanded memory to programs making EMS calls. On 286 PCs, this is done by using the BIOS Block Move (`Int 15h` Function `87h`) call to physically copy memory contents as EMS pages are mapped in. On 386/486 machines, the emulator uses the protected mode of processor operation to take advantage of the hardware paging capabilities of the chip, thereby avoiding costly memory copying operations.

On any machine there is contention for extended memory. Without VCPI, a user who runs both extended DOS applications and EMS-sensitive applications is forced to either reboot the machine between applications in order to install and deinstall the EMS emulator, or partition extended memory by configuring the EMS emulator to allocate some extended memory and leave some free. Neither approach is desirable, since both are inconvenient for the user.

On 386 and 486 PCs, additional conflicts arise out of the need of both the EMS emulator and the DOS extender to use protected mode. The only way to run an extended DOS application is to turn off the EMS emulator or deinstall it. If the emulator is turned off, the memory in the EMS memory pool cannot be used, and the computer must be rebooted to remove the emulator.

VCPI is designed to solve both the extended memory contention problems and the protected-mode conflicts. Since VCPI is 386/486-specific, we cannot use it for compatibility on 286 PCs. The reasons for this choice are: 1) protected-mode conflicts do not exist on 286s, and 2) EMS emulators are not popular on the 286, due to the overhead required for memory copying; most 286 EMS users install add-in hardware EMS boards instead.

Protected-Mode Conflicts

EMS emulators for 386- or 486-based PCs operate in protected mode and use V86 mode to run DOS and DOS-based applications. Using V86 mode allows the EMS emulator to utilize the paging capability of the processor, while still permitting standard 8086 DOS programs to execute. When an EMS emulator is installed, the processor runs primarily in V86 mode. When an interrupt occurs, the EMS emulator control software gains control in protected mode; it then chooses whether to service the interrupt in protected mode (for EMS system calls), or "reflect" the interrupt back to V86 mode for normal processing by the DOS or BIOS interrupt handler.

When the processor runs in V86 mode, it always operates at privilege level 3, the least privileged level. Software executing above privilege level 0 cannot execute certain instructions, including instructions to access the system registers (control registers, debug registers, and so on). Thus, when an EMS emulator switches the processor to V86 mode, a DOS extender cannot subsequently gain control of the machine and switch to protected mode for its own purposes. Any attempt to switch to protected mode results in a processor exception, which passes control to the EMS emulator's protected-mode exception handler.

Since an unaided switch to protected mode is not possible, extended DOS applications cannot run on a 386 or 486 PC with an EMS emulator installed. Recall that 286-specific DOS extenders can also run on 386/486 systems; they still run in protected mode, but differ from 386/486 DOS extenders by supporting 16-bit rather than 32-bit execution. While the protected mode compatibility problem is restricted to operation on 386/486 PCs, all DOS extenders, including 286-specific products, are affected.

The most important service that VCPI provides is, therefore, a means for a DOS extender to switch between V86 mode and protected mode. This service actually involves more than just switching processor modes. Since both the EMS emulator and the DOS extender provide protected-mode environments, the mode switch must also be accompanied by a switch to the appropriate environment. The DOS extender uses several VCPI calls to initialize its own protected-mode environment, making this mode and environment switching possible.

Extended Memory Allocation

Before we examine the problem of memory contention between DOS extenders and EMS emulators, let's see how DOS programs allocate extended memory.

MS-DOS is an 8086 program, and can only address memory below 1 megabyte. For this reason, no DOS services exist to allocate extended memory. Extended memory is an important resource used not only by EMS emulators and DOS extenders, but also by popular products such as RAM disk drivers, disk cache programs, and XMS drivers. Programs must mark their usage of extended memory to prevent corruption by another program attempting to use the same memory. Two common techniques exist for allocating extended memory: top-down and bottom-up.

Top-down memory allocation obtains memory from the end of extended memory, down. The program allocating extended memory hooks the BIOS Get Extended Memory Size call (`Int 15H` Function `88H`) and reduces extended memory size accordingly. Suppose that a program needs 1 megabyte of extended memory, and is executing on a machine with 3 megabytes of extended memory (4 megabytes of total memory): it installs an `Int 15H` handler that returns 2 megabytes (rather than 3) when the BIOS Get Extended Memory Size call is made. The program has effectively allocated extended memory between 3 and 4 megabytes.

Bottom-up memory allocation takes extended memory, starting at 1 megabyte, and grows up. The bottom-up technique is an older, more complicated method first used by the IBM VDISK driver. The extended memory usage is marked in two locations. A program using bottom-up allocation must hook `Int 19H`, the reboot interrupt. At a specific offset in the segment at which the `Int 19H` interrupt vector points, the program places an ASCII signature identifying the owner of the interrupt as a VDISK driver. At a second offset in the same segment is a value specifying the amount of extended memory the program uses.

The second location at which a program using bottom-up allocation marks its memory usage is a data structure called a *boot block* (remember, this method was first used by a RAM disk product). The boot block data structure is always placed in memory at 1 megabyte. Among other information, it contains a signature identifying it as a VDISK, and a value indicating its usage of extended memory.

Regardless of which technique a program uses, it must respect allocations already made by other programs. The top of extended memory must be obtained by calling the BIOS Get Extended Memory Size function. The bottom of extended memory is calculated by checking for a VDISK signature and extended memory usage in both the `Int 19H` handler and the boot block at 1 megabyte. Cautious programs print a warning message and use the larger of the two bottom-up values if they differ, or if a VDISK signature is found in one location but not in the other.

Most or all currently available EMS emulators, DOS extenders, disk cache programs, and XMS drivers use top-down extended memory allocation. Most RAM disk drivers use the bottom-up allocation technique.

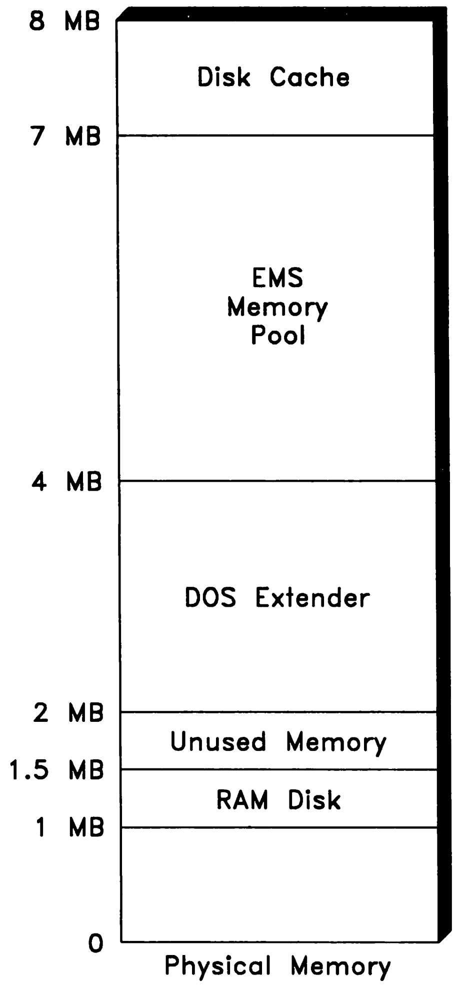
Both top-down and bottom-up allocation permit multiple programs to allocate extended memory. However, both techniques suffer from the same restriction: for each technique, only the last program to allocate extended memory can dynamically modify the amount of memory allocated. For example, if program A uses top-down allocation to take memory from 3 megabytes to 4 megabytes, and program B later uses top-down allocation to obtain memory from 2 megabytes to 3 megabytes, program A cannot increase (or decrease) its usage of extended memory, because it is no longer first in the `Int 15H` handler chain.

Figure 8-1 shows an example of memory allocation on a machine with 8 megabytes of memory and four active programs using extended memory. One program, a RAM disk, uses bottom-up allocation to obtain memory from 1 to 1.5 megabytes. Three programs have allocated extended memory with the top-down method: first (and therefore last in the `Int 15H` handler chain), a disk cache using memory from 7 to 8 megabytes; next, an EMS emulator with an EMS memory pool from 4 to 7 megabytes; and last, a DOS extender. In addition to making calls to the EMS emulator to obtain memory from the EMS memory pool, the DOS extender directly allocates extended memory between 2 and 4 megabytes. Extended memory from 1.5 to 2 megabytes, which is still unused, could be allocated by a fifth program using either top-down or bottom-up allocation. Alternatively, either the DOS extender or the RAM disk could choose to increase its memory allocation, since they are the last programs installed using their respective allocation methods. In practice, a RAM disk rarely changes its memory allocation dynamically, but a DOS extender is quite likely to do so.

With these techniques for allocating extended memory in mind, we can consider contention problems between EMS emulators and DOS extenders. Since the available allocation methods are not dynamic, an EMS emulator grabs all of its memory when it is installed. If the user configures the emulator to take all available extended memory (the default for most products), none is then available for a subsequent extended DOS application. A DOS extender can, of course, allocate

EMS memory; but there is no service in the EMS interface to get the physical address of allocated memory—information that is needed by the DOS extender.

Figure 8-1: Extended memory allocation.



VCPI, therefore, includes services both to allocate EMS memory and obtain its physical address directly, and to obtain physical addresses for memory allocated through the standard EMS interface. Since a user can configure an EMS emulator to take all, some, or none of the available extended memory, a DOS extender should be capable of both allocating extended memory directly and using EMS memory (and, of course, other sources of memory such as conventional DOS memory and XMS memory). The DOS extender can then obtain the maximum amount of available memory regardless of how the user configures the EMS emulator.

The VCPI Interface

VCPI consists of a set of services provided by the EMS emulator (the *server*) and used by the DOS extender (the *client*). Technically, a VCPI server is any program that provides both the EMS 4.0 and the VCPI interface, and a VCPI client is any program that makes VCPI calls. In this chapter, we use the term *server* interchangeably with EMS emulator, and *client* with DOS extender.

Table 8-2 lists the calls provided by a VCPI server. Some calls are provided in V86 mode only, some in both V86 and protected modes, and one (a mode switch) only in protected mode. V86-mode calls are made via the standard EMS `Int 67H` interface, with an EMS function number of `DEH` in register AH, and a VCPI function number in register AL. The client makes VCPI calls in protected mode with a far procedure call to an entry point in the server. The server's protected-mode entry point is obtained during the interface initialization process. As with V86-mode calls, protected-mode VCPI calls set register AH to `DEH` and pass a VCPI function number in AL.

Table 8-2: VCPI call summary.

Function Number	Modes Available	Function Name	Description
Initialization			
00h	V86	VCPI Presence Detection	Used to detect the presence of an EMS emulator that provides the VCPI interface.
01h	V86	Get Protected Mode Interface	Allows the server to initialize portions of the client's system tables to facilitate communication between client and server in protected mode.
02h	V86	Get Maximum Physical Memory Address	Returns the physical address of the highest 4K memory page the server will ever allocate.

Function Number	Modes Available	Function Name	Description
Memory Allocation			
03h	V86, Prot	Get Number of Free 4K Pages	Determines how much memory is available.
04h	V86, Prot	Allocate a 4K Page	Allocates a page of memory.
05h	V86, Prot	Free a 4K Page	Frees a previously allocated page of memory.
06h	V86	Get Physical Address of 4K Page in First Megabyte	Translates a linear address in the server's memory space to a physical address.
System Register Access			
07h	V86	Read CR0	Obtains value in the CR0 register.
08h	V86	Read Debug Registers	Obtains values in the six debug registers.
09h	V86	Load Debug Registers	Loads specified values into the debug registers.
Interrupt Controller Management			
0Ah	V86	Get 8259A Interrupt Vector Mappings	Obtains the interrupt vectors used to signal hardware interrupts.
0Bh	V86	Set 8259A Interrupt Vector Mappings	Notifies the server that the client has reprogrammed the interrupt controllers to use different interrupt vectors.
Mode Switching			
0Ch	V86	Switch From V86 Mode to Protected Mode	Allows the client to switch to protected mode and set up its own environment.
0Ch	Prot	Switch From Protected Mode to V86 Mode	Allows the client to switch back to execution in V86 mode in the server's environment.

The calls are grouped into five distinct categories. Only the memory allocation calls, and the call to switch from protected mode to V86 mode, are available when the client is executing in protected mode. The same VCPI function number, `0CH`, is used for the mode switch call in both directions. This duplicate use of a function number is not ambiguous, since only two processor modes—V86 and protected—are supported by VCPI.

Interface Initialization

When a DOS extender begins execution, it detects the presence of a VCPI server by: 1) checking for the presence of an EMS driver, 2) allocating an EMS page to turn the EMS emulator on if it is off, thus causing it to switch the processor to V86 mode, and 3) making the VCPI presence detection call (Function 00H). The EMS emulator must be turned on to enable the VCPI interface.

During initialization, the client makes the Get Protected Mode Interface call (Function 01H). This call sets up the conditions that make it possible for the server to switch between its own environment and that of the client, and for the client to make calls to the server from protected mode. The Get Interface call also returns the server's protected-mode entry point for VCPI calls. We will examine this call in more detail in the section *Inside VCPI*, later in this chapter.

The client may optionally choose to use Get Maximum Physical Memory Address (Function 02H) during initialization. This call is provided for clients who need to know in advance the physical address of the highest memory page that will ever be allocated, so they can initialize their memory management data structures.

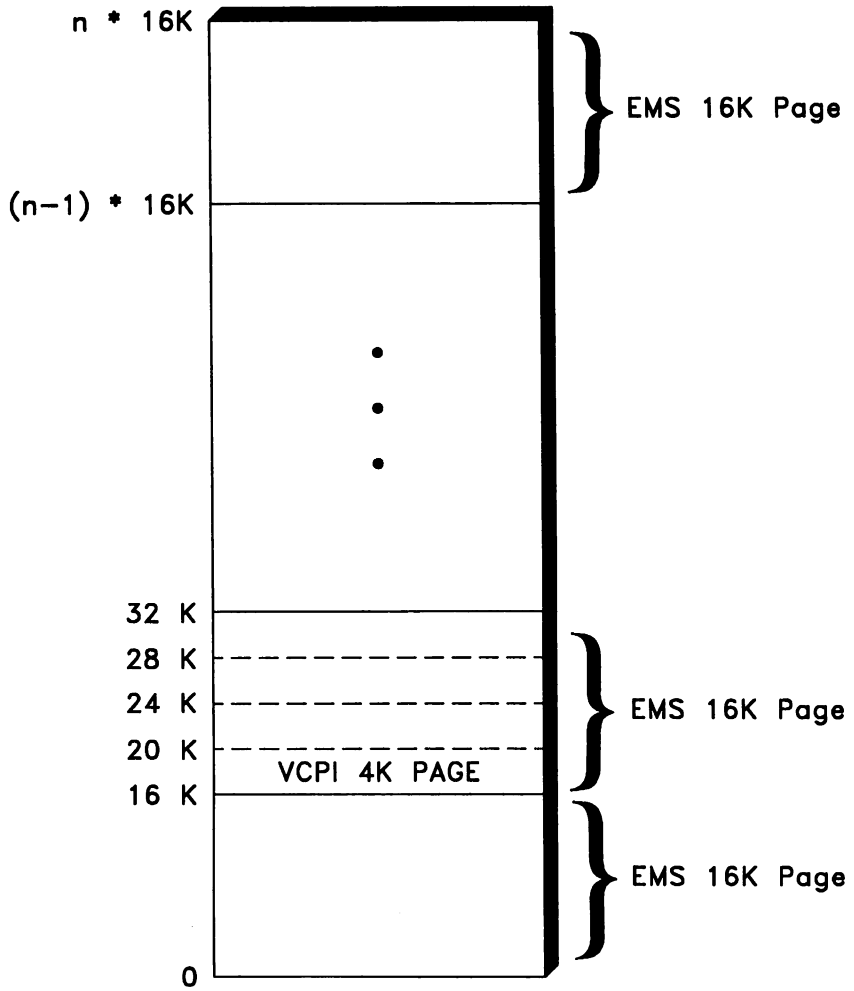
Memory Allocation

Memory is allocated in units of 4K pages under VCPI. Notice this is the page size used by the 386/486 hardware paging unit, not the 16K page size used by the EMS interface. The EMS memory pool is a contiguous chunk of extended memory allocated by the emulator when it is installed. The memory pool can be thought of as an array of 16K EMS pages (see Figure 8-2). Each 16K page is further subdivided into four 4K pages (which are aligned on 4K physical address boundaries, a requirement of the processor's paging unit). Memory can be allocated in the EMS memory pool in two ways: 1) by making EMS calls to allocate memory in 16K units, or 2) by making VCPI calls to allocate memory in 4K units.

A VCPI client can choose to use either or both of the above allocation methods to obtain memory from the EMS emulator. The VCPI allocation calls are provided for clients who prefer the page size units used by the hardware, or who want to allocate memory in protected mode without the overhead of mode switching. Memory allocated with VCPI calls is required to be in the processor's memory address space (excluding, for example, hardware expanded memory boards).

VCPI includes calls to allocate (Function 04H) and free (Function 05H) a single 4K page of physical memory. The number of free 4K pages available can be obtained via Function 03H. These calls can be made in both V86 and protected mode. Allocated pages are identified by their physical address rather than an arbitrary "handle"; clients must deal in physical memory addresses since they set up their own protected-mode environment, including page tables mapping allocated memory.

Figure 8-2: The EMS memory pool.



The Get Physical Address of 4K Page in First Megabyte call (Function 06H) is provided for clients using the EMS interface to allocate memory. The client allocates an EMS page and maps it into the EMS page frame in the usual way. It then calls Function 06H to obtain the physical address of each of the four 4K pages that make up the EMS page. This call gives the client the ability to allocate EMS memory with standard EMS calls. Unlike the VCPI allocation calls, both the EMS calls and the Get Physical Address VCPI call can be made only in V86 mode, since, of course, the server's environment must be in effect for the server to translate a linear address to a physical address.

Figure 8-3 compares the two methods for allocating physical pages. It shows a physical address space, in which all extended memory is used for the EMS memory pool. On the left side of the diagram, the VCPI Allocate 4K Page call allocates a single 4K memory page and obtains its physical address. On the right side, EMS calls allocate a 16K EMS page and maps it into the EMS page frame in the DOS memory space. Then the VCPI Get Physical Address function obtains the physical address of one of the 4K pages in the 16K EMS page.

System Register Access

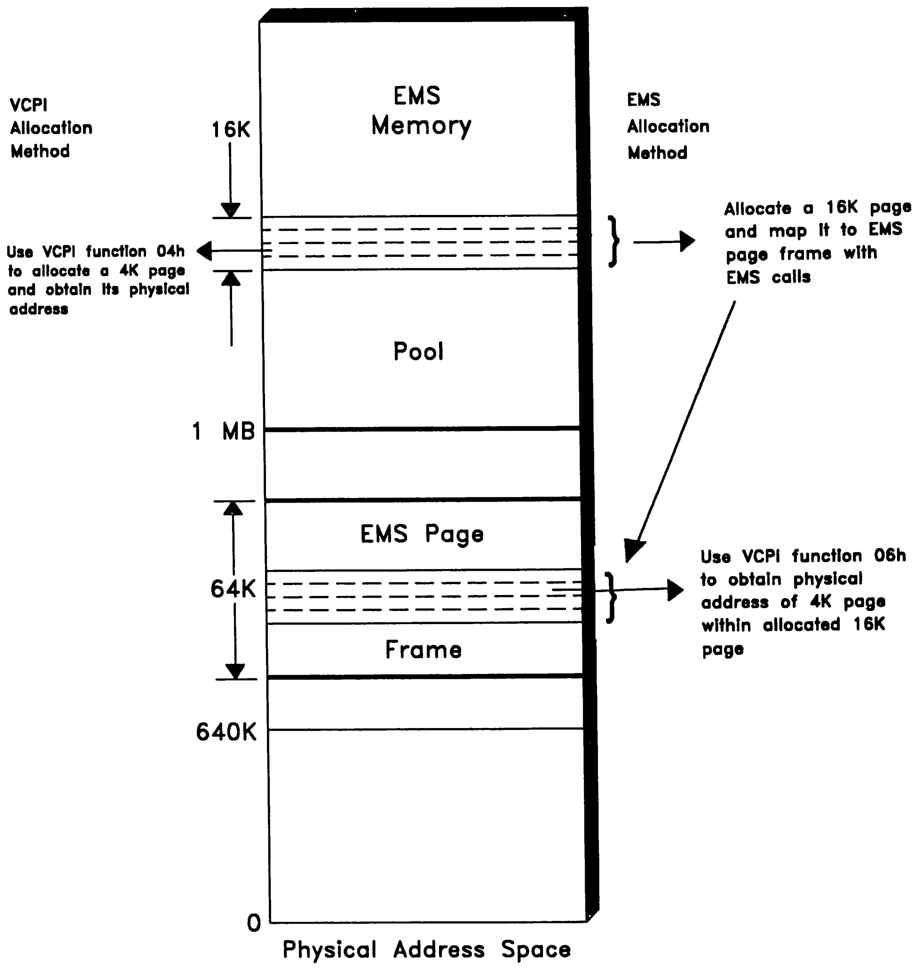
The system registers of the 386/486 processors (CR0, CR2, CR3, GDTR, LDTR, IDTR, TR, the test registers, and the debug registers) cannot be directly stored or loaded in V86 mode; any attempt to do so results in a processor exception. VCPI clients do not need access to most of these registers in V86 mode; several of them are useful only in protected mode, and are loaded as part of the environment switch that is performed when a mode switch is requested. The interface does support V86 mode access to some of the system registers.

Read CR0 (Function 07H) is provided to give the client the ability to examine the CR0 register, which is not normally accessible in V86 mode. This call exists primarily for historical reasons; many clients do not use it, and the low 16 bits of CR0 can, in any case, be read directly in V86 mode with the SMSW instruction.

The Read Debug Registers and Load Debug Registers calls (Functions 08H and 09h) allow the client to read and write the processor debug registers in V86 mode. The debug registers are used to set up to four code breakpoints and/or data watchpoints in application programs, and to determine the source of a debug exception. Most 386-specific debuggers use these debug registers.

Since DOS extenders support application programs that run partially in protected mode and partially in real mode, debuggers for extended DOS applications must be capable of setting breakpoints and watchpoints in real-mode code as well as protected-mode code. While a client could always access the debug registers directly by first switching to protected mode, it is often more convenient to read and write debug registers in V86 mode when debugging real-mode code.

Figure 8-3: Allocating EMS memory.


VCPI Allocation Method
16K
EMS Memory
EMS Allocation Method
Allocate a 16K page and map it to EMS page frame with EMS calls
Use VCPI function 04h to allocate a 4K page and obtain its physical address
Pool
1 MB
EMS Page
64K
Frame
Use VCPI function 06h to obtain physical address of 4K page within allocated 16K page
640K
0
Physical Address Space

Interrupt Controller Management

DOS extenders must handle all interrupts that occur when their protected-mode environment is in effect. Hardware interrupts are often processed slightly differently than software interrupts, so the DOS extender typically needs to know which interrupt vectors are used for hardware interrupts. If the standard DOS hardware interrupt vectors are still in effect, some DOS extenders relocate IRQ0–IRQ7 to make it easier to distinguish between hardware interrupts and processor exceptions.

The Get and Set 8259A Interrupt Vector Mappings calls (Functions 0AH and 0Bh) give the client the ability to determine which interrupt vectors are used for hardware interrupts. In real mode, vectors 08h–0FH are used for IRQ0–IRQ7, and vectors 70h–77H are used for IRQ8–IRQ15. Because interrupts 08H–0FH are also used for processor exceptions, some protected-mode control programs reprogram the interrupt controllers to use different interrupt vectors for hardware interrupts. The Get 8259A Mappings call allows the client to determine which vectors are used for hardware interrupts.

If the standard vector mappings are still in effect, the client is permitted to reprogram the interrupt controllers and make the Set 8259A Mappings call to inform the server of the new vector mappings. If the interrupt controllers have already been reprogrammed, the client is not permitted to modify the mappings.

Mode Switching

A DOS extender must perform mode switches to V86 mode and back again every time an interrupt occurs, whether it is a hardware interrupt, a processor exception, or a software interrupt such as a DOS or BIOS system call. Mode switches are made with the VCPI calls Switch From V86 Mode to Protected Mode and Switch From Protected Mode to V86 Mode. These calls use the same function number (you can think of Function 0CH as the *mode switch* function).

The mode switch calls do more than just switch processor modes; they also switch between the server's and the client's environments. This switching of environments is what really makes VCPI work; it allows both server and client to behave as if they "own" the machine.

Scenario of VCPI Use

As an example of how a DOS extender uses the VCPI interface, let's take a look at some of the actions performed to execute the simple protected-mode assembly language program shown below. The program just makes two DOS calls: one to print the message "Hello world!" and the second to terminate.

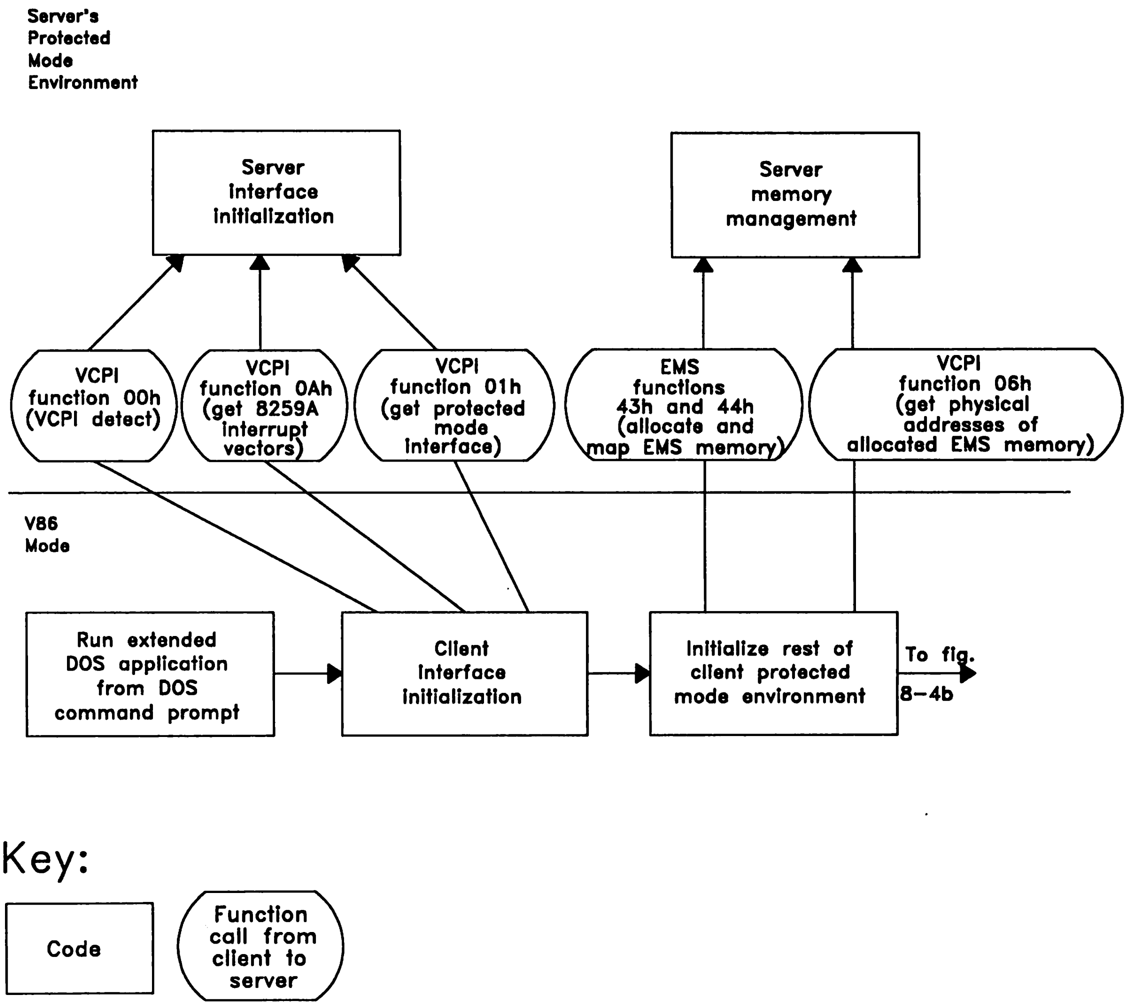
```
;
; This is a complete protected mode program. It is built with the
; Phar Lap assembler and linker commands:
;       386asm hello.asm
;       386link hello.obj
;
; It is "bound" with Phar Lap's 386|DOS-Extender as follows:
;       bind386 run386b.exe hello.exp
;
; This creates a HELLO.EXE file, which is run in the standard DOS fashion:
;       hello
;
        assume  cs:cseg,ds:dseg
cseg    segment byte public use32 'code'
start   proc    near
        mov     ah,09h                  ; Print hello world
        lea     edx,hellomsg
        int     21h

        mov     ax,4C00h                ; exit to DOS
        int     21h
start   endp
cseg    ends

dseg    segment dword public use32 'data'
hellomsg db     'Hello world!',0Dh,0Ah,'$'
dseg    ends

sseg    segment dword stack use32 'stack'
        db      8192 dup (?)
sseg    ends

        end start
```

Figure 8-4a shows the initialization phase of the DOS extender. The client determines that the VCPI interface is present by using Function 00H as part of the detection sequence described earlier. It then calls Function 0AH (Get 8259A Interrupt Vector Mappings) to obtain the interrupt vectors used to signal hardware interrupts, so it can set up appropriate protected-mode interrupt handlers for each of the 256 interrupts.

Next, Get Protected Mode Interface (Function 01H) is called to initialize the page table mappings for up to the first 4 megabytes of the client's linear address space, and to get the protected-mode entry point in the EMS emulator. The DOS extender then completes the setup of the page table mappings for the rest of its protected-mode linear address space, and initializes its system data structures

(GDT, LDT, IDT, and so on). In this example, the memory for the system tables is allocated from the EMS memory pool, to minimize use of conventional DOS memory. This is done by allocating and mapping in EMS pages, and using VCPI Function 06H to obtain physical addresses for the 4K pages within each allocated EMS page. (The DOS extender is unlikely to allocate memory with the Allocate 4K Page call during initialization because it is not yet prepared to switch to protected mode, so it has no way to initialize memory for which it has only a physical address.)

Figure 8-4a: DOS extender initialization phase.


Server's Protected Mode Environment
Server interface initialization
Server memory management
VCPI function 00h (VCPI detect)
VCPI function 0Ah (get 8259A interrupt vectors)
VCPI function 01h (get protected mode interface)
EMS functions 43h and 44h (allocate and map EMS memory)
VCPI function 06h (get physical addresses of allocated EMS memory)
V86 Mode
Run extended DOS application from DOS command prompt
Client interface initialization
Initialize rest of client protected mode environment
To fig. 8-4b
Key:
Code
Function call from client to server

With the above processing completed, the DOS extender is now prepared to switch to protected mode and load the application program. Figure 8-4b shows the program loading phase. First, the DOS extender uses VCPI Function 0CH to switch to protected mode. It then allocates memory in which to load the program by making repeated calls to Allocate 4K Page (Function 04H).

The application program is then loaded into the allocated memory. The file I/O to read the program in from disk is done with DOS Int 21H function calls. The software interrupt causes control to pass to the DOS extender's protected-mode Int 21H handler. This handler switches to V86 mode and reissues the interrupt to DOS, taking care of buffering data appropriately.

Figure 8-4b: Program loading phase.

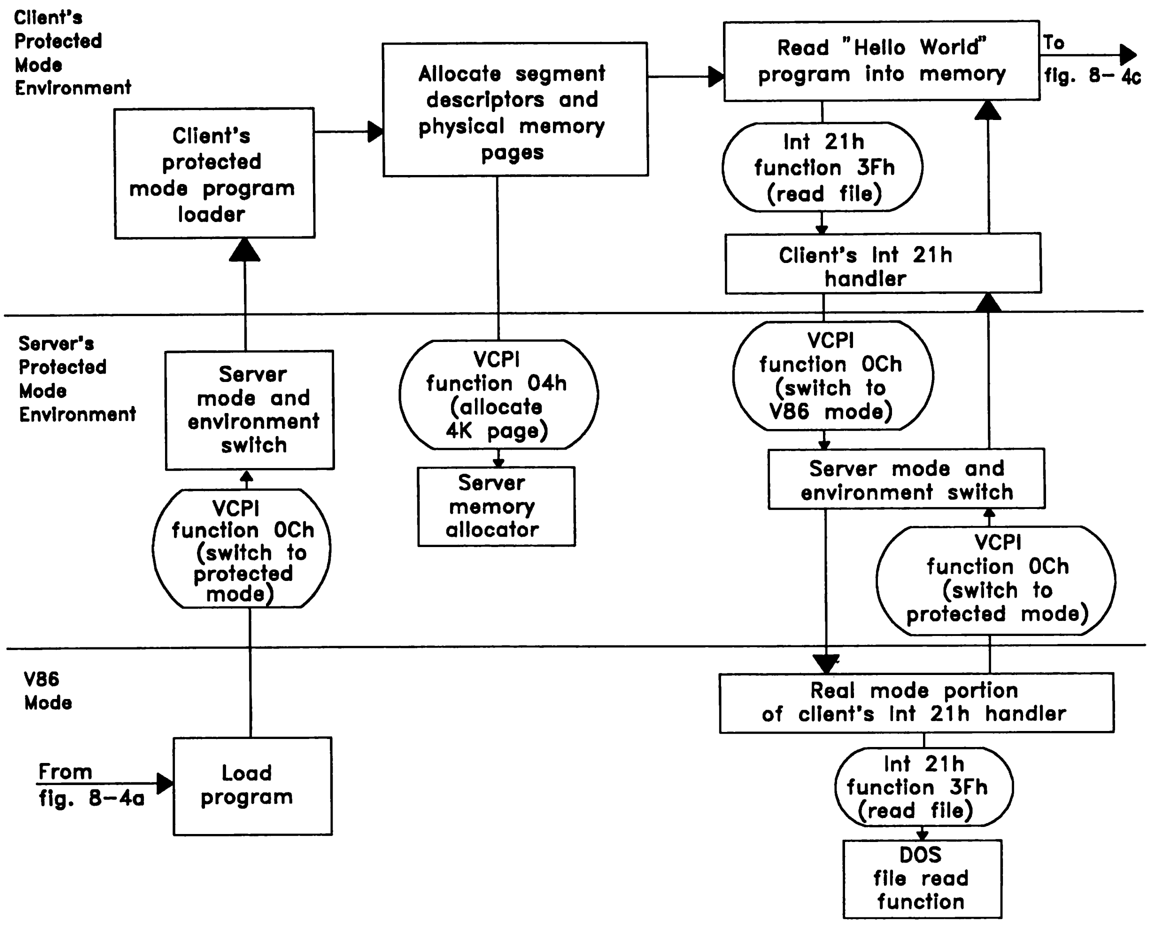
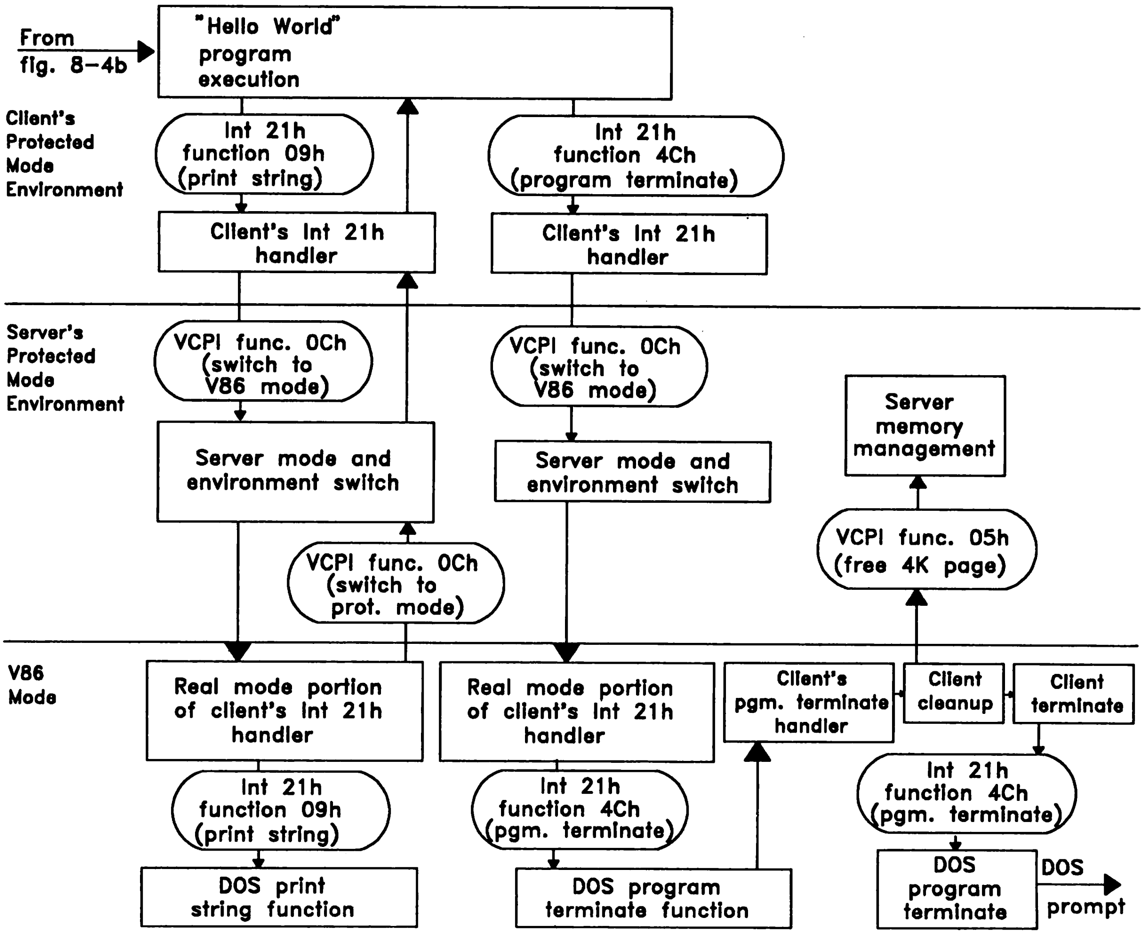

Client's Protected Mode Environment
Client's protected mode program loader
Allocate segment descriptors and physical memory pages
Read "Hello World" program into memory
To fig. 8-4c
Int 21h function 3Fh (read file)
Client's Int 21h handler
Server's Protected Mode Environment
Server mode and environment switch
VCPI function 04h (allocate 4K page)
Server memory allocator
VCPI function 0Ch (switch to V86 mode)
Server mode and environment switch
VCPI function 0Ch (switch to protected mode)
VCPI function 0Ch (switch to protected mode)
V86 Mode
From fig. 8-4a
Load program
Real mode portion of client's Int 21h handler
Int 21h function 3Fh (read file)
DOS file read function

Figure 8-4c shows the application program execution phase. The DOS extender passes control to the application's entry point in protected mode. The application issues an `Int 21H` to print the "Hello world!" string. This invokes the DOS extender protected-mode `Int 21H` handler, which copies the string to a buffer located in the first 640K, where it can be addressed by DOS. The handler then switches back to V86 mode and reissues the `Int 21H` so DOS can process the call. After the DOS function completes, control returns to the application, in protected mode.

Figure 8-4c: Program execution phase.


From fig. 8-4b
"Hello World" program execution
Client's Protected Mode Environment
Int 21h function 09h (print string)
Int 21h function 4Ch (program terminate)
Client's Int 21h handler
Client's Int 21h handler
Server's Protected Mode Environment
VCPI func. 0Ch (switch to V86 mode)
VCPI func. 0Ch (switch to V86 mode)
Server memory management
Server mode and environment switch
Server mode and environment switch
VCPI func. 05h (free 4K page)
VCPI func. 0Ch (switch to prot. mode)
V86 Mode
Real mode portion of client's Int 21h handler
Real mode portion of client's Int 21h handler
Client's pgm. terminate handler
Client cleanup
Client terminate
Int 21h function 09h (print string)
Int 21h function 4Ch (pgm. terminate)
Int 21h function 4Ch (pgm. terminate)
DOS print string function
DOS program terminate function
DOS program terminate
DOS prompt

Finally, the application makes the DOS program terminate call, which is handled in the same fashion as the earlier print string call. The DOS extender gets control from DOS in V86 mode after the application program terminates. The client then frees physical memory that was allocated to the application program. VCPI memory is returned to the server with VCPI Function 05H, conventional memory is freed by calling DOS, and so on. After completing its cleanup, the DOS extender calls the DOS terminate function to exit back to the DOS command prompt.

Inside VCPI

Both EMS emulators and DOS extenders, when run independently, "own" the machine; that is, they control the system tables (GDT, LDT, IDT, TSS, and page tables) required for operation when the processor is in protected mode. Each program expects to see its own protected-mode environment (*environment* refers to the set of system tables listed above) when it is active.

Under VCPI, the server always has control while the processor is executing in V86 mode. When the client calls the server to switch from V86 to protected mode, the server must give control to the client in the client's protected-mode environment. Conversely, when the client requests a switch back to V86 mode, the server must switch back to its own protected-mode environment before passing control to the client in V86 mode.

Environment Correspondence

Switching between two protected-mode environments is a tricky operation. Some correspondence between the two environments is required for the switch to be possible. The Get Protected Mode Interface call (Function 01H) allows the server to initialize two key aspects of the client's environment: the page table mappings for up to the first 4 megabytes of the client's linear address space, and descriptor table entries for up to three segments in the client's GDT.

The server initializes some or all of the client's 0th page table (linear addresses from 0 to 4 megabytes). The interface requires the server to duplicate the first megabyte of its own address space in the client's page table. This linear address space duplication provides the basis for communication between the server and client in protected mode. It also gives the client the ability to access the V86 one-megabyte address space from protected mode when the client's environment is active.

The three segments the server creates in the client's GDT are used for communication between client and server when the client's environment is active. The first segment must be used as a code segment. The second and third may be used in any way the server desires; typically, at least one is used as a data segment. The server locates all three segments in the linear address region shared by the server and the client, since the server has no knowledge of the rest of the client's linear address space.

Address Space Organization

Figure 8-5 demonstrates one possible memory configuration in a VCPI environment. Keep in mind that this example is a simplified diagram and does not accurately depict memory organization under actual EMS emulators or DOS extenders; it leaves out information that does not relate directly to operation under VCPI. This diagram can help us understand how the relationships between the server's and client's environments make VCPI work.

The physical address space in the example is 4 megabytes in size. The first megabyte has the usual DOS configuration: 640K of RAM memory, with the BIOS ROM and video memory located above 640K. The entire 3 megabytes of extended memory is allocated by the EMS emulator (with the top-down allocation method) as a memory pool for EMS memory.

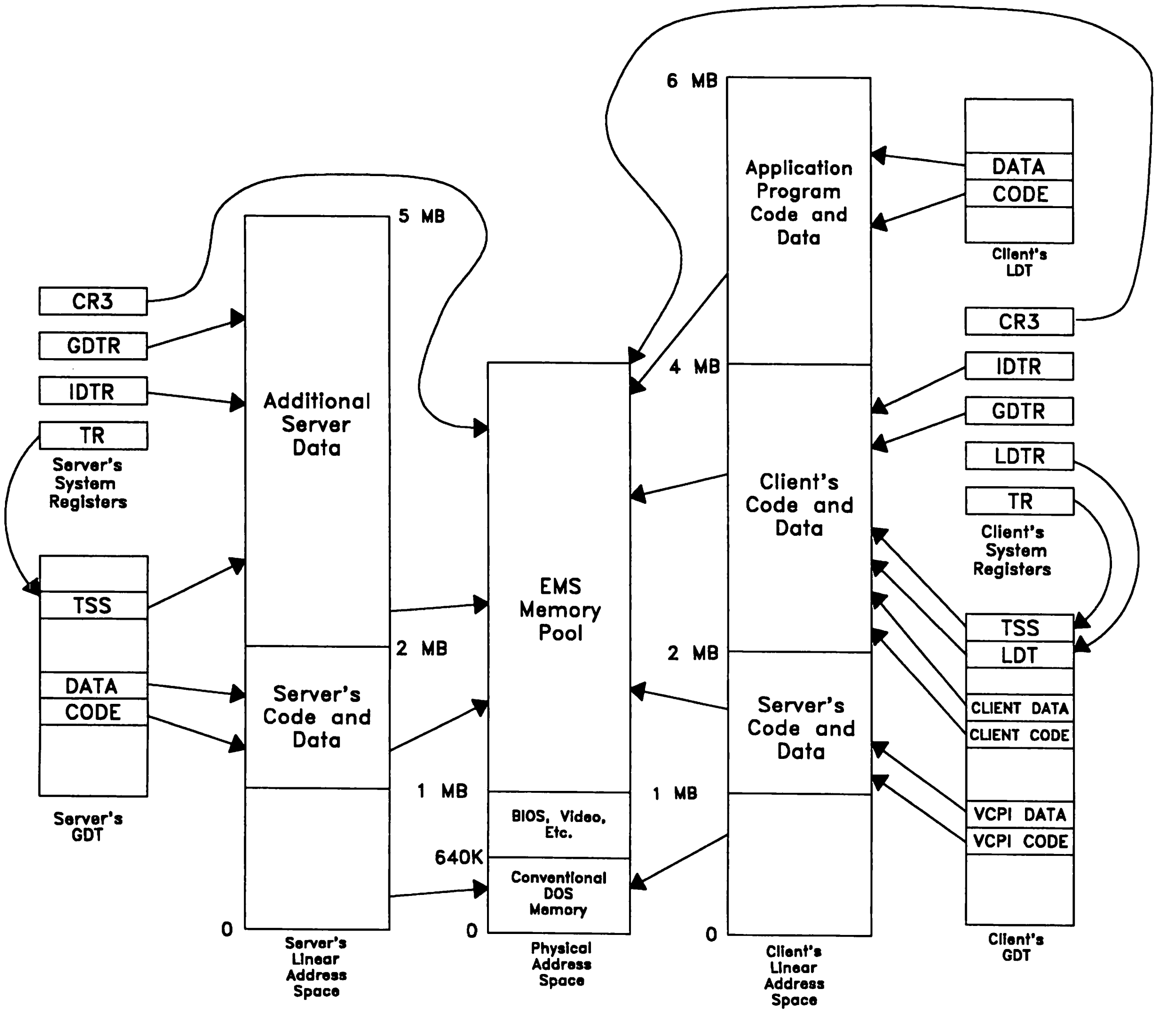
At any given moment, either the server's or the client's address space is in effect. As we saw in Chapter 1, the page tables are used to map the linear address space into the physical address space. Since the CR3 register contains the physical address of the page directory, modifying CR3 selects a linear address space by switching to a different set of page tables.

The linear address space set up by the EMS emulator is shown in the left half of the diagram. The first megabyte is given the identity mapping: linear addresses equal physical addresses. Linear addresses up to 2 megabytes are used to map the server's code and data segments. Addresses from 2 to 5 megabytes are used for additional data, such as a memory management area and the system tables (GDT, IDT, and TSS). In this example, the server does not use an LDT. In addition to a system segment for the TSS, the GDT contains the server's code and data segments. The GDT itself is, of course, located in the linear address space; it is shown as a separate structure in the figure to minimize the tangle of arrows.

All of the linear region containing the server's code and data, including system tables, is mapped to physical memory in the EMS memory pool. The server's page tables, mapped by `CR3` directly in physical memory rather than in the linear address space, are also placed in the EMS memory pool.

The client's linear address space is on the right side of the diagram. As required by the interface, the client maps linear addresses from 0 to 1 megabytes exactly as they are mapped in the server's address space. In this example, the server has initialized the client's page table such that the two linear address spaces are identical up to 2 megabytes. The server's code and data segments are located above 1 megabyte in the server's address space, and are mapped identically in both environments.

Figure 8-5: Example of environment relationships.



In this example, the DOS extender uses linear addresses from 2 megabytes to 4 megabytes to map its own code and data segments, including its GDT, LDT, IDT, and TSS. Linear addresses from 4 megabytes to 6 megabytes contain the protected-mode application's code and data. The GDT contains six segments: the system segments for the client's LDT and TSS, the client's code and data segments, and the VCPI code and data segments. The server initializes the VCPI segments in the client's GDT when the client makes the Get Protected Mode Interface call.

The linear region containing the client's code, data, and system tables, and the application's code and data, is mapped to physical memory in the EMS memory pool. The client's page tables are likewise allocated in the EMS memory pool.

It is instructive to consider how a real-world configuration might complicate the situation shown in the example. As you read the following (by no means comprehensive) list of likely differences, try to visualize how each item changes Figure 8-5.

- The 64K of memory directly above 1 megabyte is mapped by the server to physical memory from 0 to 64K; this is for compatibility with DOS programs, which rely on the address space wrapping at 1 megabyte that occurs when address line 20 is disabled, as is normally the case under DOS.
- A disk cache and a RAM disk are installed; they each use some extended memory, leaving less available for the EMS memory pool.
- The EMS emulator is configured to "backfill" unused address space in the region between 640K and 1 megabyte. This extra memory is used to load TSRs and device drivers, keeping them out of the 640K DOS memory. Some linear regions in the server's address space between 640K and 1 megabyte are now mapped to physical memory in the EMS memory pool, rather than given the identity mapping.
- The EMS page frame, also located between 640K and 1 megabyte, points to some allocated and mapped EMS pages in the memory pool.
- The client's GDT contains several code and data segments, at least two of which map V86 code and data located in the first 640K of the linear address space.
- The client's LDT has segments to map the application program's PSP and DOS environment variables in the first 640K of the linear address space.

- The physical memory allocated for the application's code and data segments (client linear addresses from 4 to 6 megabytes) comes partially from the EMS memory pool, and partially from physical memory below 640K.
- The DESQview386 multitasker is installed. The first 150K of the linear address space has the identity mapping, but linear addresses between 150K and 640K are mapped to physical memory in the EMS memory pool because more than one virtual machine is active.

Switching Between Environments

Switching from the server's environment to the client's (or back again) requires the following operations:

- Reloading CR3, the page directory base register, which switches from the server's linear address space to the client's.
- Reloading GDTR, the register containing the GDT linear base address and limit.
- Reloading LDTR, the register containing the LDT segment selector. This selector references an LDT system segment descriptor in the client's GDT.
- Reloading TR, the register containing the segment selector for the TSS system segment descriptor in the GDT. Before loading TR, the "task busy" bit in the TSS segment descriptor must be cleared, since the instruction that loads TR sets the task busy bit, and generates a processor exception if the busy bit was previously set.
- Reloading IDTR, the register containing the IDT linear base address and limit.

The server has limited flexibility in its choice of the order in which to perform these operations. CR3 must be loaded first, since all the other register values have relevance only in the context of the client's linear address space. LDTR and TR must be loaded after loading GDTR, since they reference segment descriptors in the GDT. IDTR may be loaded at any point after loading CR3. Note, however, that interrupts must remain disabled for the duration of this operation, since the segment registers all contain selectors referencing the server's GDT—so interrupt handlers (which save and restore the contents of segment registers) must not be permitted to execute.

The VCPI call to switch from V86 mode to protected mode passes to the server the linear address of a data structure containing the client's system register values, and the protected-mode entry point in the client (see the VCPI call descriptions). This VCPI call is implemented in the example below. The environment switch is accomplished with just a few instructions, but the logic is complex enough to be worth a close look.

```
;
; This protected-mode code fragment switches from the server's environment
; to the client's environment and transfers control to the client. This is
; part of the processing performed when the client makes the VCPI call
; Switch From V86 Mode to Protected Mode.
; The code can be assembled with the Phar Lap assembler as follows:
;       386asm switch.asm
;
; The following requirements must be met by the server before calling this
; routine to transfer control to the client:
;  1. This routine and the current stack must both be located within the
;     linear address region that is mapped identically by both server
;     and client.
;  2. DS must be loaded with a selector for a writable data segment whose
;     base address is zero and whose limit is 4 GB.
;  3. Interrupts must be disabled. This routine is required to transfer
;     control to the client with interrupts still disabled.
;  4. On entry to this routine, all general registers contain the values
;     they had when the client issued the Switch to Protected Mode call.
;     This routine can modify only EAX, ESI, and the segment registers.
;

        .prot           ; enable assembly of privileged instructions
        assume  cs:cseg
cseg segment byte public use32 'code'
switch    proc    far

;
; Format of the data structure to which ESI points on entry to this routine
;
SWSTR   struc
        SW_CR3          dd      ?       ; client's CR3 value
        SW_GDTOFFS      dd      ?       ; offset of client's GDTR value
        SW_IDTOFFS      dd      ?       ; offset of client's IDTR value
        SW_LDTR         dw      ?       ; client's LDTR value
        SW_TR           dw      ?       ; client's TR value
        SW_EIP          dd      ?       ; entry point in client
        SW_CS           dw      ?
SWSTR   ends

        mov     eax,[esi].SW_CR3        ; load CR3 to switch to client's linear
        mov     cr3,eax                 ; address space
        mov     eax,[esi].SW_GDTOFFS    ; set up client's GDT
        lgdte   pword ptr [eax]
        mov     eax,[esi].SW_IDTOFFS    ; set up client's IDT
        lidte   pword ptr [eax]
        lldt    [esi].SW_LDTR           ; set up client's LDT
        mov     eax,[esi].SW_GDTOFFS    ; set EAX = linear base address of
        mov     eax,2[eax]              ; client's GDT
        push    ebx                     ; index to client's TSS descriptor
        mov     bx,[esi].SW_TR
        and     ebx,0FFF8h
        add     eax,ebx
        pop     ebx
        and     byte ptr 5[eax],not 2   ; clear task busy bit in TSS descr.
        ltr     [esi].SW_TR             ; set up client's TSS
        jmp     pword ptr [esi].SW_EIP  ; jump to client's entry point

switch  endp
cseg    ends
        end
```

The first operation the code performs is reloading CR3, which sets up the client's page tables, thus switching to the client's linear address space (the right-hand side of Figure 8-5). Since an entry requirement of the routine is a DS segment with a base address of zero and a limit of 4 gigabytes, the linear address in ESI is used as a direct offset in the (assumed) DS data segment. For this operation to succeed, the code and any data it references (including its stack) must be located in the linear region (0 - 2 megabytes in our example) mapped identically in the server's and client's address spaces. Recall that the VCPI call Switch to Protected Mode requires the data structure passed in by the client to reside in the first megabyte; this guarantees it can be accessed in either address space.

After switching to the client's address space, and before setting up the rest of the client's environment, we must be careful to avoid any operation that references a linear address above the linear region shared by the server and client. This includes loading segment registers; the GDTR still contains the address of the server's GDT, which is no longer accessible because it is not in the shared linear region. Likewise, interrupts cannot be permitted, since the IDT is also no longer mapped.

You may be wondering how the processor can execute code or access data if the GDT is unavailable. The answer lies in the way the processor addresses segments in protected mode. The segment descriptor in the GDT or LDT is read-only when a segment register is loaded with a selector; the processor keeps the segment's linear base address, limit, and access rights in an internal cache register, so it doesn't have to constantly reference the GDT or LDT. Thus, if we don't attempt to reload a segment register, we can continue to use existing server segments, provided they are located in the shared linear address region.

The second and third operations load the GDTR and IDTR registers. These registers contain the linear base address and limit of the client's GDT and IDT. It may seem reasonable to enable interrupts at this point, since the client's page tables, GDT, and IDT are all in effect, but in fact, this isn't possible. To see why, consider how a typical interrupt handler operates. It saves registers it plans to modify—including segment registers—on the stack, processes the interrupt, restores the saved register values, and returns. The trouble arises when the handler reloads segment registers with saved values: the saved values are selectors that reference descriptors in the server's GDT, but the current GDT is the client's GDT.

Next, the LDTR register is loaded. Since the LDTR register contains a segment selector that references a system segment descriptor in the client's GDT, it must be loaded after loading GDTR.

Next, the TR register is loaded. This register contains a segment selector referencing a TSS descriptor in the GDT. The LTR instruction sets the task busy bit in the TSS descriptor, and requires that the busy bit be clear before the instruction is executed. The server must therefore clear the busy bit in the descriptor before loading TR. The tricky part of this operation arises because the client's GDT is not located in the shared linear space. The server, however, has the linear base address of the client's GDT available—it is part of the 6-byte value loaded into GDTR. This requirement to clear the task busy bit is the reason the code needs a DS segment with a limit of 4 gigabytes—the server has no control over where the client locates its GDT in the linear address space.

Finally, the code transfers control by jumping to the client's protected-mode entry point. For reasons noted above, the client must reload all segment registers, including setting up its own stack, before re-enabling interrupts.

Memory Allocation in Protected Mode

Apart from the call to switch back to V86 mode, the only VCPI calls the client can make from protected mode are memory allocation calls. These calls are provided in protected mode for program performance. In fact, performance and the convenience of 4K page size units are the only reasons the Allocate 4K Page and Free 4K Page calls are provided at all, since the client can also obtain memory by making an EMS allocate call, mapping EMS 16K pages into the EMS page frame, and using VCPI Function 06h to obtain physical addresses of the allocated memory.

When the client makes a VCPI call in protected mode, the server gets control with the client's environment in effect. The only knowledge the server has of the environment is the portion of the linear address space the server initialized when the client made the Get Interface call, and the three segment descriptors the server set up in the client's GDT. Depending on the architecture of the server program, this may be sufficient to service the VCPI request. If not, the server must switch to its own protected-mode environment, service the call, and then switch back to the client's environment to return the result to the client.

VCPI Calls

The VCPI calls are shown in Table 8-3, ordered by function number. Each call description includes the input and output parameters for the call and a brief summary of its functionality. Except where noted, all calls can be made only in V86 mode. More complete call details can be found in the VCPI specification.

Table 8-3: VCPI calls.

Function	Input Parameter	Output Parameter
VCPI Presence Detection	AX = DE00h	If VCPI present: AH = 0 BL = VCPI minor version number, in binary BH = VCPI major version number, in binary If VCPI not present: AH = nonzero
Note: The presence detection call is made after checking for the presence of an EMS emulator and allocating one EMS page to make sure the EMS driver is turned on and has switched the processor to V86 mode.		
Get Protected Mode Interface	AX = DE01h ES:DI = pointer to client's 0th page table DS:SI = pointer to three descriptor table entries in the client's GDT	AH = 0 DI = advanced to first uninitialized page table entry in client's page table EBX = offset of server's entry point in protected mode code segment

Function	Input Parameter	Output Parameter

Note: When the client makes the Get Interface call, the server initializes a portion of the client's 0th page table, which maps linear addresses from 0 to 4 megabytes in the client's address space. The server is required to map linear addresses from 0 to 1 megabyte exactly as they are mapped in the server's address space. The server also initializes up to three descriptor table entries in the client's GDT. The first of these three descriptors is required to be a code segment; the server returns its entry point in this code segment, to be used for making VCPI calls from protected mode. This segment and address space initialization lays the groundwork that makes switching between server and client environments, and communication in protected mode, possible.

Function	Input Parameter	Output Parameter
Get Maximum Physical Memory Address	AX = DE02h	AH = 0 EDX = physical address of highest 4K page server will allocate

Note: The client uses this call during initialization to determine the largest physical page address the server can allocate. Some clients use the information for memory management; it is convenient to set up a segment mapping the entire physical address space, so physical pages can be read and written directly.

Function	Input Parameter	Output Parameter

Get Number of Free 4K Pages	AX = DE03h	AH = 0
		EDX = number of free 4K pages

Note: The Get Number of Free Pages call returns the current number of unallocated 4K pages in the server's memory pool. This call can be made in both protected mode and V86 mode.

Function	Input Parameter	Output Parameter

Allocate a 4K Page	AX = DE04h	If success:
		AH = 0
		EDX = physical address of 4K page
		If failure:
		AH = nonzero
		EDX = unspecified

Note: The client makes this call to allocate a page of memory. This call can be made in both protected mode and V86 mode.

Function	Input Parameter	Output Parameter

Free a 4K Page	AX = DE05h
	EDX = physical address of previously allocated 4K page
		If success:
		AH = 0
		If failure:
		AH = nonzero

Note: The Free Page call is used to free a page of memory allocated via Function 04H. The client is required to free all allocated memory before exiting. This call must not be used to free memory allocated through the EMS interface. The Free Page call can be made in both protected mode and V86 mode.

Function	Input Parameter	Output Parameter

Get Physical Address of 4K Page in First Megabyte	AX = DE06h
	CX = page number (linear address of page shifted right 12 bits)
		If success:
		AH = 0
		EDX = physical address of 4K page
		If invalid page number:
		AH = nonzero

Note: The client uses this call primarily to obtain the physical addresses of memory allocated through the EMS interface and mapped into the EMS page frame. If the allocated memory is to be used in protected mode, the client needs the physical address so it can map the page in its own page tables.

Function	Input Parameter	Output Parameter

Read CR0	AX = DE07H	AH = 0
		EBX = CR0 value

Note: The Read CR0 call returns the current value in the CR0 register.		
Read Debug Registers	AX = DE08h ES:DI = pointer to array of 8 DWORDs, DR0 first in array, DR4 and DR5 not used	AH = 0
Note: The client makes this call to obtain the current debug register values.		
Load Debug Registers	AX = DE09h ES:DI = pointer to array of 8 DWORDs, DR0 first in array, DR4 and DR5 not used	AH = 0
Note: The client uses this call to load the debug registers with the specified values.		

Get 8259A Interrupt Vector Mappings	AX = DE0Ah	AH = 0
		BX = interrupt vector for IRQ0
		CX = interrupt vector for IRQ8

Note: This call returns the interrupt vectors generated by the 8259A interrupt controllers when a hardware interrupt occurs.		

Set 8259A Interrupt Vector Mappings	AX = DE0Bh
	BX = interrupt vector for IRQ0
	CX = interrupt vector for IRQ8
	Interrupts disabled
		AH = 0

Note: The client uses this call to inform the server that it has reprogrammed the interrupt controllers to use different interrupt vectors. The client is required to disable interrupts before reprogramming the interrupt controllers, and to make this call before re-enabling interrupts. The client is permitted to perform this operation only if the standard DOS hardware interrupt vectors (08h - 0FH for IRQ0 - IRQ7, 70H - 77H for IRQ8 - IRQ15) are in effect before the client reprograms the interrupt controller. If the client reprograms the interrupt controllers, it is required to restore the original vector mappings and inform the server with this call before terminating.		
Switch From V86 Mode to Protected Mode	AX = DE0Ch ESI = linear address below 1 megabyte of data structure specified below Interrupts disabled	GDTR, IDTR, LDTR, TR loaded with client's values Interrupts disabled Control transferred to specified FAR entry point in client EAX, ESI, DS, ES, FS, GS unspecified
Note: The data structure at which ESI points on input to this call is organized as shown in Table 8-4. The server switches to the client's environment by loading the processor's system registers with the client's values, and then transfers control to the specified entry point in the client. Care is required when switching environments; the section *Inside VCPI,* early in this chapter, contains an example implementation of this call.		

Function	Input Parameter	Output Parameter
Switch From Protected Mode to V86 Mode	AX = DE0Ch Interrupts disabled DS = segment with a base address of 0 and a limit at least as large as the address space initialized by the server when the Get Protected Mode Interface call was made. SS:ESP must be in linear memory below 1 megabyte, and points to the data structure described in Table 8-5.	GDTR, IDTR, LDTR, TR loaded with server's values Interrupts disabled SS:ESP, DS, ES, FS, GS loaded with specified values Control transferred to specified FAR entry point in client EAX unspecified

Note: The data structure on the top of the stack is organized as shown in Table 8-5.
The server switches to its own protected-mode environment by loading the processor's system registers with its own values, and then passes control to the client in V86 mode with the segment registers loaded with the values in the data structure on the stack. The data structure is organized so that the server can switch to V86 mode and load the client's registers with a single IRETD instruction. The server must initialize the EFLAGS value, since it controls the IOPL setting in V86 mode. This call can be made only from protected mode.

Table 8-4: Data structure for switch to protected mode.

Offset	Description
00H	(DWORD) client's CR3 value
04H	(DWORD) linear address below 1 megabyte of 6-byte variable containing client's GDTR value
08H	(DWORD) linear address below 1 megabyte of 6-byte variable containing client's IDTR value
0CH	(WORD) client's LDTR value
0EH	(WORD) client's TR value
10H	(PWORD) protected-mode entry point in client

Table 8-5: Data structure for switch to V86 mode.

Offset	Description
00H	(QWORD) FAR return address from procedure call (not used)
08H	(DWORD) client's V86-mode EIP value
0CH	(DWORD) client's V86-mode CS value
10H	(DWORD) reserved for EFLAGS value
14H	(DWORD) client's V86-mode ESP value
18H	(DWORD) client's V86-mode SS value
1CH	(DWORD) client's V86-mode ES value

Offset	Description
20H	(DWORD) client's V86-mode DS value
24H	(DWORD) client's V86-mode FS value
28H	(DWORD) client's V86-mode GS value

Summary

When programs use resources not managed by the operating system under which they run, potential conflicts arise. In the case of DOS extenders and EMS emulators, the non-DOS resources used by both are extended memory and the protected mode of processor execution. The VCPI interface is designed to resolve the contention for these two resources.

VCPI can be thought of as an extension of EMS 4.0 for the 386/486, and consists of a set of services used by a DOS extender. The EMS emulator, which is installed first (usually when the PC starts up), is the provider of the VCPI interface. When an extended DOS application is run, the VCPI-aware DOS extender detects the presence of the interface and uses VCPI calls to allow it to switch to protected mode and allocate memory owned by the EMS emulator.

Most popular EMS emulators, and most DOS extenders used to create protected-mode DOS applications, now support VCPI. PC users can install EMS emulators on their 386 and 486 machines and run extended DOS applications without worrying about conflicts—in fact, without even being aware of any potential resource contention problems or of the existence of VCPI. While VCPI solves only one specific compatibility problem, it is a valuable technology because of the large (and growing) popularity of both EMS emulators and extended DOS application programs.

Chapter 9

The DOS Protected-Mode Interface (DPMI)

Ray Duncan

The DOS Protected Mode Interface (DPMI) has as its primary goal the "safe" execution of DOS extender protected-mode applications within DOS-based multitasking environments. It addresses the problems of contention for system resources such as extended memory, the descriptor tables that control protected-mode addressing, the processor's debug registers, the interrupt subsystem, and the video display adapter that inevitably arise when two or more high-performance protected-mode applications are executing in the absence of a true protected-mode operating system.

In some respects, a DPMI "server" program is similar to a VCPI server, in that it provides mode-switching and extended memory management services to client programs. But unlike a VCPI server, a DPMI server runs at a higher privilege level than its clients using the hardware to enforce a "kernel/user" protection model. This allows a DPMI server to support demand paged virtual memory and maintain full control over client programs' access to the hardware via device virtualization. Furthermore, the DPMI's functions for memory and interrupt management are much more general than those in the VCPI. The VCPI was originally designed for ease of implementation in EMS emulators such as Qualitas's 386-to-the-MAX, and requires a 386 or 486 CPU, whereas the DPMI may also be implemented on an 80286.

The prototype of the DPMI was developed by Microsoft for Windows version 3.0, with input from Lotus and Rational Systems, as part of a general effort to enhance Windows' performance by allowing true Windows applications to run in extended memory. Microsoft then decided to involve the other manufacturers of EMS emulators, DOS extenders, and other multitasking environments to make sure DPMI could serve as an adequate platform for all DOS-based software that uses protected mode and extended memory. In February 1990, an ad-hoc committee composed of Microsoft, Intel, Borland, Eclipse, IBM, IGC, Lotus, Phar Lap, Quarterdeck, and Rational Systems took charge of the DPMI specification and began the process of recasting it as a vendor-independent, industry-wide standard.

As this book goes to press, the DPMI is still in a period of transition, and it is likely that the interface will undergo many additional changes before the DPMI reaches its final form. Consequently, this chapter will only describe the DPMI at a functional level, and will not include detailed programming information.

The DPMI Interface

The function calls supported by the DPMI fall into seven general categories, as shown on pages 400-402. Let's look at each of these groups a little more closely.

The *LDT Management* functions allow a program to manipulate its own Local Descriptor Table. It can allocate and free selectors, inspect or modify the descriptors that are associated with allocated selectors, map real-mode addresses onto protected-mode selectors, and obtain a read/write data selector ("alias") for an executable selector. Functions are also provided that allow a program to lock down segments for performance reasons, or to prevent segments that contain interrupt handlers from being paged out to disk.

Interestingly, access to descriptors is provided at two levels of abstraction. A program can construct a descriptor in a buffer, setting all the various fields explicitly, then copy the selector into the descriptor table as a unit. Alternatively, a program can call individual functions to set a descriptor's base address, limit, and access-rights byte in separate operations. Note that access to the Global Descriptor Table (GDT) is not provided, so that the DPMI server can protect itself from protected-mode applications and isolate those applications from each other to any extent it wishes.

The *DOS Memory Management* functions basically provide a protected-mode interface to the real-mode MS-DOS `Int 21H` Functions `48H` (Allocate Memory Block), `49H` (Free Memory Block), and `4AH` (Resize Memory Block). Using these functions, a protected-mode program can obtain memory below the 640K limit, which it can use to exchange data with MS-DOS itself, TSRs, ROM BIOS device drivers, and other real-mode programs that are not capable of reaching data in extended memory or interpreting protected-mode addresses.

The *Extended Memory Management* functions are roughly the equivalent of the DOS Memory Management functions, but allocate, resize, and release blocks of physical memory above the 1-megabyte boundary. The functions are low-level in that they do not allocate selectors or build descriptors for the extended memory blocks; the program must allocate selectors and map the memory onto the selectors with additional DPMI calls. When the DPMI server is a 386/486 control program with paging enabled, the extended memory blocks are always allocated in multiples of 4K.

The *Page Management* functions allow memory to be locked or unlocked for swapping on a page by page basis, in terms of the memory's linear address. These functions provide control of the system's virtual memory management at a finer granularity than that afforded by the segment lock and unlock functions already mentioned under *LDT Management* services. If the DPMI server is running on a 286, or is running on a 386/486 but does not support demand paging, the *Page Management* group of functions have no effect.

The first six *Interrupt Management* functions listed on page 401 allow protected mode programs to intercept software or hardware interrupts that occur in real mode or protected mode, or install handlers for processor exceptions and faults (such as divide by zero and overflow) that are issued on interrupts `0` through `1FH`. Normally, the handlers for real-mode interrupts and faults must be located in locked memory below the 640K boundary, although it is possible to transfer control at interrupt time to a handler in extended memory by means of a real-mode Call-Back (see below). The last three *Interrupt Management* functions allow a process to enable or disable its own servicing of hardware interrupts, without affecting the interrupt status of the system as a whole. The DPMI server implements this by trapping the process's execution of CLI and STI, and maintaining a "virtual interrupt" flag on a per-process basis.

The *Translation Services* provide a mechanism for cross-mode procedure calling. In particular, a protected-mode program can transfer control to a real-mode routine by either a simulated far call or a simulated interrupt. Parameters may be passed to the real-mode routine in registers (by initializing a structure similar to that used by the C `int86()` function), on the stack, or both. When a 32-bit protected-mode program calls a 16-bit real-mode procedure, the DPMI function will perform the appropriate copying and truncation of stack parameters, and even supply the real-mode stack itself if necessary. The Translation Services also allow a protected-mode program to declare an entry point (called a Real-Mode Call-Back) which can be invoked by a real-mode program. This would be useful, for example, in the case of the Microsoft Mouse driver, which can be configured to notify an application program whenever the mouse is moved or a button is pressed or released.

The *Miscellaneous Services* include a function to get the version of the DPMI supported by the DPMI server, and a function to convert an arbitrary physical memory address into a linear (paged) memory address. The version function should be called by a DPMI client during its initialization, to make sure that all the DPMI services that it needs are really available. The address conversion function allows a protected-mode program to reach a memory-mapped I/O device whose physical address lies above the 1-megabyte boundary. If the DPMI server is running on a 286 machine, or on a 386/486 machine with paging disabled, the returned linear address is the same as the physical address. In any case, once the linear address is obtained, the program must still allocate a selector and map the linear address to the selector.

The following list contains the DPMI services exported to protected-mode DOS extender applications. The grouping of services shown here is for convenience of discussion in this chapter, and is not the same as the grouping found in the DPMI Specification.

LDT Management Services

0000H	Allocate LDT Selector
0001H	Free LDT Selector
0002H	Map Real Mode Segment to Selector
0003H	Get Next Selector Increment Value
0004H	Lock Selector Memory
0005H	Unlock Selector Memory
0006H	Get Selector Base Address
0007H	Set Selector Base Address
0008H	Set Selector Limit
0009H	Set Selector Access Rights
000AH	Create Code Segment Alias Selector
000BH	Get Descriptor
000CH	Set Descriptor
000DH	Allocate Specific LDT Selector

DOS Memory Management Services

0100H	Allocate DOS Memory Block
0101H	Free DOS Memory Block
0102H	Resize DOS Memory Block

Extended Memory Management Services

0500H	Get Free Memory Information
0501H	Allocate Memory Block
0502H	Free Memory Block
0503H	Resize Memory Block

Page Management Services

0600H	Lock Linear Region
0601H	Unlock Linear Region
0602H	Mark Real Mode Region as Pageable
0603H	Relock Real Mode Region

Interrupt Management Services

0200H	Get Real Mode Interrupt Vector
0201H	Set Real Mode Interrupt Vector
0202H	Get Processor Exception Handler Vector
0203H	Set Processor Exception Handler Vector
0204H	Get Protected Mode Interrupt Vector
0205H	Set Protected Mode Interrupt Vector
0900H	Get and Disable Virtual Interrupt State
0901H	Get and Enable Virtual Interrupt State
0902H	Get Virtual Interrupt State

Translation Services

0300H	Simulate Real Mode Interrupt
0301H	Call Real Mode Procedure with Far Return Frame
0302H	Call Real Mode Procedure with Interrupt Return Frame
0303H	Allocate Real Mode Call-Back Address
0304H	Free Real Mode Call-Back Address
0305H	Get State Save Addresses
0306H	Get Raw CPU Mode Switch Addresses

Miscellaneous Services

0400H	Get DPMI Version
0800H	Physical Address Mapping

Using the DPMI

A crucial point to understand with regard to the DPMI is that—like the VCPI—it is an interface intended for use by DOS extenders only; under normal circumstances, DPMI functions would never be called directly by an application program. Proper use of the DPMI functions requires an understanding of protected-mode machine resources that lies far outside the scope of normal application programming. It is the DOS extender's job to build upon the DPMI functions to provide a protected-mode application with memory, addressability, and "transparent" access to such DOS and ROM BIOS functions as the application might need.

DPMI functions are invoked by executing an `Int 31H`, with a function number in register AX. Other parameters typically are also passed in registers, although a few DPMI functions require data in structures that are pointed to by DS:SI or ES:DI. DPMI functions indicate success by clearing the carry flag, or an error by setting the carry flag; other results are usually returned in registers.

The scenario for loading and initialization of a DOS extender under the DPMI is similar to the sequence described in Chapter 8 for DOS extenders running under the VCPI. A DOS extender is initially loaded in real mode on 286 machines, or in virtual 86 mode on 386/486 machines. Before using any `Int 31H` functions, it must check for the existence of the DPMI by executing an `Int 2FH` with the value `1687H` in AX. If a DPMI server is present, the following values are returned:

■ AX	= 0
■ BX	= Flags (bit 0 = 1 if 32-bit programs supported)
■ CL	= Processor type (02H=80286, 03H=80386, 04H=80486)
■ DH	= DPMI major version number
■ DL	= DPMI minor version number
■ SI	= number of paragraphs required for DOS extender private data
■ ES:DI	= DPMI mode switch entry point

The DOS extender can then call the entry point whose address was returned in ES:DI to switch the CPU from real mode or virtual 86 mode into protected mode. Upon return from the mode switch call, the CPU is in protected mode; CS, DS, and SS have been loaded with valid selectors for 64K segments that map to the same physical memory as the original real-mode values in CS, DS, and SS; ES contains a selector that points to the program's PSP with a 256-byte segment limit; FS and GS contain zero (the null selector) if the host machine is a 80386 or 80486; and all other registers are preserved.

Having gotten this far, the DOS extender is in a position to use DPMI services to construct the protected-mode environment that will be used by the actual application program. It must allocate extended memory segments to hold the application's code, data, and stacks, and allocate selectors that will be used by the application to execute in and/or address those memory segments. It can then read the application's code and data from disk into the newly created segments, and perform any necessary relocations. If the DOS extender chooses to use memory below the 640K boundary, it can mark that memory as pageable to reduce the total demand for physical memory. In most cases, the DOS extender will also supply the application with selectors that address vital structures such as the PSP, environment block, ROM BIOS data area, and video refresh buffers.

Before transferring control to the application, the DOS extender must install its own handlers for any software interrupts that will be executed by the application (such as the MS-DOS `Int 21H` or the ROM BIOS video driver `Int 10H`). The DOS extender's processing of such software interrupts may range from very simple to very complex. For example, the `Int 21H` character input and output functions can mostly be passed right through to DOS; the DOS extender needs to do little else than switch to real mode before calling DOS, and switch back to protected mode upon DOS's return. Other functions, such as file input and output, may also require the DOS extender to translate addresses or copy data between the application's buffers and other buffers below the 640K boundary for DOS's benefit. And some `Int 21H` functions, such as the DOS memory management functions, must be replaced by the DOS extender entirely.

In addition to whatever handling the DOS extender provides for `Int 21H` functions requested by the application program, the DPMI also filters all `Int 21H` calls. When the DOS extender or its application executes an `Int 21H` with `AH=4CH` (the standard DOS termination function), the DPMI traps the interrupt and releases all of the program's protected-mode resources. The DPMI then "passes down" the termination function to the real-mode owner of `Int 21H`, so that DOS can clean up the program's real-mode resources such as file and device handles and memory blocks below the 640K boundary.

Summary

The DPMI is a second-generation descendant of the VCPI, in that it provides a means for protected mode DOS extender applications to run cooperatively in the presence of 386/486 memory managers, multitaskers, and control programs. It resolves some problems related to device virtualization, inter-task protection, and demand paging that were technically difficult or impossible to address in the VCPI because of the VCPI's original design.

In one way, the DPMI can be viewed as a substrate for the multitasking, protected-mode, easily programmed version of DOS that did not materialize in OS/2. If the DPMI is eventually subsumed into DOS itself, the future of desktop computing may take quite a different course than the one currently envisioned by either UNIX or OS/2 partisans. In the meantime, inclusion of the DPMI in Windows 3.0 assures it will rapidly achieve a very large installed base, and makes it likely that the DPMI will become an attractive target for vendors of DOS extenders and other 286/386/486-aware software in a fairly short period of time.

Chapter 10

Multitasking and DOS Extenders

Robert Moote

Multitaskers that run on top of DOS, such as DESQview 386 and Windows 3.0, are available for all types of PCs, from 8088-based machines on up. On PCs that are based on the newer 80286, 80386, and 80486 processors, these multitaskers can run in protected mode and take advantage of extended memory, device virtualization, and V86 mode to run more applications at a time and to prevent the applications from interfering with each other. But when multitaskers use these advanced chip features, they are assuming much of the role of a true protected-mode operating system, and they can come into conflict with DOS extenders which also rely on these advanced CPU characteristics.

Some conflicts between multitaskers and DOS extender applications can be resolved through the VCPI interface that was discussed in Chapter 8. For example, VCPI-aware DOS extenders can run under DESQview 386. However, because VCPI does not permit DESQview to perform hardware virtualization, there are some limitations on running extended DOS applications under DESQview.

Although Windows 3.0 does not support VCPI, extended DOS applications can be run under Windows when it is operating in real mode, or in standard mode with some limitations. Current DOS extenders cannot run under Windows 3.0 in its enhanced mode.

The DPMI interface provided in Windows 3.0 theoretically allows a DOS extender to attain full compatibility with Windows. The existence of this interface (described in Chapter 9), and the fact that industry leaders have established a committee to endorse the interface, gives some hope that future releases of DOS extender products will be able to cooperate with all multitaskers which take full advantage of the capabilities of the 80386 and 80486 CPUs. But the benefits of DPMI are as yet unproven, and there are currently no DOS extenders that are capable of using it.

In this chapter, we will examine in some detail the compatibility problems that arise when a DOS extender runs under a multitasker, and describe the kind of interface that is necessary for multitaskers to provide complete hardware isolation to DOS extenders. We will also look at the interim solution used by DESQview 386 for execution of DOS extenders: a combination of VCPI support and certain restrictions on the DOS extender's actions.

Multitasking on 80286-Based PCs

On 286 machines, a multitasker is obligated to run DOS programs in real mode because there is no V86 mode on a 286 processor. Extended or expanded memory is used for swapping tasks that are not currently active. True hardware virtualization is not possible, because real-mode programs run at privilege level 0, the most privileged level of the processor.

Since the multitasker executes applications in real mode, DOS extenders that are loaded under the multitasker are free to take control of the machine and run in protected mode. There are potential problems with contention for extended memory and relocation of hardware interrupt vectors, but these problems can usually be solved at the user level by configuring the multitasker and/or the extended DOS application appropriately. And on 286 machines, hardware virtualization for DOS extenders is no more (or less) difficult than it is for any DOS application.

Existing DOS extenders are compatible with most 286-specific multitaskers, including Windows 3.0 operating in its real mode or standard mode, and the problems that do exist on 286 machines are a subset of the more general compatibility issues faced on 386/486 machines. Consequently, the remainder of this chapter is concerned only with compatibility on the 386 and 486. The memory contention and hardware interrupt problems just mentioned are discussed below in the context of the 386/486; the possible solutions identified can also be applied to 286-specific products.

Sources of Incompatibilities on 386- and 486-Based PCs

The conflicts between EMS emulators and DOS extenders examined in Chapter 8 —use of V86 mode and memory sharing—are also a source of compatibility problems between DOS extenders and multitaskers. In addition, a new class of problems is introduced due to the nature of the services provided by multitaskers. Since a multitasker runs several programs simultaneously, each of which thinks it owns the machine, the multitasker must virtualize the hardware and prevent programs from interfering with each other. The most obvious shared hardware resource is the display, but other examples are the keyboard, mouse, and communication ports. For the multitasker to be able to virtualize hardware, it must be in control of the machine at all times.

Recall that the VCPI interface works by allowing each of the cooperating programs (the EMS emulator and the DOS extender) to set up its own environment while it executes. This environment separation is what makes adding VCPI support to existing products (which are written assuming they own the machine) relatively straightforward. Conversely, the multitasker's requirement that its environment always be active is the reason implementation of an interface permitting DOS extenders and multitaskers to cooperate is a difficult task.

V86 Mode

Most multitaskers, including DESQview 386 and Windows 3.0 enhanced mode, use the V86 processor mode because of its technical advantages for multitasking real-mode applications. As noted in Chapter 8, this prevents a DOS extender (or any other program that wishes to switch to protected mode) from gaining control of the machine. The cooperation of the multitasker is necessary if the DOS extender is to be able to switch to protected mode. The multitasker can choose to provide the VCPI interface, but that solution leaves other problems unaddressed, as we shall see.

Memory Sharing

The problem of extended memory allocation, discussed in Chapter 8, also exists in a multitasking environment. However, with a multitasker the problem is compounded by the requirement for multiple programs to obtain memory concurrently. The multitasker needs the ability to limit the memory consumption of any one program, so that a single program cannot prevent all others from executing by grabbing all the memory. Thus, in a multitasking environment, a DOS extender should only obtain memory from the multitasker.

A second memory issue is implementation of virtual memory. Both DOS extenders and multitaskers commonly provide virtual memory services, but it is inefficient to have two levels of memory virtualization active simultaneously. For good performance, a DOS extender and a multitasker should cooperate such that one, but not both, provides virtual memory services.

Hardware Virtualization

386-specific multitaskers use V86 mode and the processor's I/O permission bit map to virtualize the hardware when running standard real-mode DOS programs. Because programs running in V86 mode always run at the least privileged level of the processor, the multitasker can arrange to get control when any program executes an I/O instruction, attempts to access a system register, or attempts to read from or write to a portion of the memory space (such as the video memory) which the multitasker wants to protect. The multitasker can let each program think it has direct control of the hardware, but still prevent simultaneously executing programs from interfering with each other.

When a DOS extender runs in V86 mode, it runs at privilege level 3 like any other program, so it poses no special problems to a multitasker. But DOS extenders and their applications run primarily in protected mode, not in V86 mode. Furthermore, the DOS extender expects to be in control of the machine when it is executing in protected mode—it has its own GDT, LDT, IDT, and page tables. If the multitasker allows the DOS extender to execute in this way—for instance, if the multitasker provides VCPI—then the multitasker is unable to virtualize direct hardware access by the DOS extender or the protected-mode application program. For complete compatibility it is therefore desirable that the DOS extender not set up its own protected-mode environment, but rather function within the environment created by the multitasker.

A second problem for virtualization is that DOS extenders always run at privilege level 0 (the most privileged level) in protected mode, and often also run the application program at level 0. While it is possible for the multitasker to provide memory address space virtualization to level 0 programs, it cannot prevent level 0 code from performing direct hardware I/O or accessing system registers.

Therefore, it is also necessary that both the DOS extender and the application program run at a less privileged level than the multitasker's kernel.

Hardware Interrupts

The original IBM PC design (based on the 8088) used interrupt levels `08h-0Fh` for the eight hardware interrupts `IRQ0` through `IRQ7`, in spite of the fact that Intel had reserved interrupt levels `0-1FH` for processor exceptions. Those interrupts were, in fact, used for processor exceptions on the later 286, 386, and 486 processors. When the 286-based AT was introduced, eight more hardware interrupts were added (`IRQ8-IRQ15`); these were assigned to interrupt levels `70h-77h`, a choice which does not cause any additional compatibility problems. Many multitaskers and DOS extenders reprogram the interrupt controller chip to reassign `IRQ0-IRQ7` so there is no ambiguity about the source of an interrupt. Multitaskers often also relocate `IRQ8-IRQ15` to aid in virtualizing hardware interrupts.

It is clearly not acceptable for both the multitasker and a DOS extender running under the multitasker to reprogram the interrupt controller. The two programs must cooperate so that only the multitasker relocates interrupts, and the multitasker must inform the DOS extender which interrupt vectors are used for hardware interrupts.

Possible Solutions

For multitaskers and DOS extenders to coexist in a 386/486-based environment, they must cooperate using an explicit software interface.

A partial solution that permits limited compatibility is implementation of the VCPI interface in the multitasker. A DOS extender can run in such an environment, but the multitasker is unable to virtualize the hardware for extended DOS applications. This can be an acceptable restriction for many users' requirements. The VCPI solution is provided by DESQview 386; we will examine it more closely below.

A better solution that solves all the compatibility problems entails the definition of a new interface, to be provided by multitaskers and used by DOS extenders. One such interface is DPMI, recently defined by Microsoft in collaboration with several software and hardware vendors, and included in Windows 3.0.

The salient characteristic of DPMI, or of any interface designed to solve DOS extender/multitasker compatibility, is the requirement for the DOS extender to run both in the multitasker's environment, and at a lesser privilege level (higher ring number) than the multitasker. This gives the multitasker the ability to virtualize hardware accesses performed by an extended DOS application.

Services provided by DPMI include mode switching, memory management, segment management, hardware and software interrupt control, and management of processor exceptions. Ideally, the interface should include sufficient services to allow DOS extenders to continue to support all currently provided functionality for protected-mode application programs. DPMI comes reasonably close to meeting this ideal.

DPMI is not currently supported by DOS extenders because the interface has only recently been documented, and considerable work is required to re-engineer existing DOS extender products to use DPMI. All current DOS extender products set up their own protected-mode environment under the assumption that they can control the machine in protected mode. Recall that the VCPI interface was designed to permit that assumption to remain valid, so that the interface would not be difficult to implement. Changing that assumption so that a DOS extender can control the machine and provide its own protected- mode environment when executed under plain-vanilla DOS, but use the DPMI interface and run at a lesser privilege level when a multitasker is installed, will require a substantial rewrite of the DOS extender.

DPMI offers the potential for full compatibility between multitaskers and extended DOS applications. However, due to the magnitude of the conversion task, it is unclear how soon DOS extender developers will include DPMI support in their products.

DESQview 386 and DOS Extenders

DESQview 386 is capable of multitasking extended DOS applications because it cooperates with the VCPI interface. The QEMM-386 EMS emulator distributed with DESQview provides VCPI, and DESQview uses VCPI to inform the DOS extender where hardware interrupts are mapped. DESQview also preserves the 32-bit general registers and the debug registers across a context switch, obviously a necessity for the multitasking of 32-bit programs.

The goal of allowing extended DOS applications to run under DESQview is achieved by this choice. Recall there are two multitasking compatibility problems not adequately addressed by VCPI: memory sharing between applications, and hardware virtualization. The memory sharing problem is solved by requiring the DOS extender to follow specific memory allocation conventions. Hardware virtualization is not possible, and this results in some restrictions on how extended DOS applications can be run under DESQview.

Memory Allocation

A typical DOS program attempts to allocate all available memory, under the assumption that it is the only active program on the system. Protected-mode extended DOS applications are no exception. DESQview simultaneously runs multiple memory-hungry programs by limiting the amount of memory available to each program. It does this by intercepting DOS and EMS memory allocation calls to make it appear to the application as if less memory exists than is actually the case.

Protected-mode VCPI calls to allocate and free memory cannot be filtered by DESQview since they do not require a transition through V86 mode where the DESQview environment is active. An extended DOS application could potentially bypass the memory restrictions placed by a user on a DESQview window and allocate all available EMS memory, thereby preventing applications in other windows from obtaining enough memory to run.

There are two solutions to this problem. The first is for the DOS extender to allocate all EMS memory via EMS calls in V86 mode rather than with VCPI calls in protected mode. Since memory allocation is now performed in V86 mode, DESQview is able to intercept the calls and limit memory consumption.

The second solution requires the voluntary cooperation of the DOS extender. During initialization, the DOS extender makes the EMS Get Number of Pages call (`Int 67H` Function `42H`). If DESQview is not present, this call returns the total amount of free memory in the EMS memory pool. If the program is running under DESQview, the returned value reflects the user-imposed memory limitations for that window. The DOS extender then deliberately limits its allocation of VCPI memory to the amount returned by the EMS call.

Both methods work, and all DOS extender products that support VCPI implement one of these two solutions. While neither approach is elegant, either achieves the practical goal of solving the memory contention problem.

Hardware Virtualization

DESQview is unable to prevent an extended DOS application from directly accessing hardware devices. The most commonly used hardware resource is the display; many programs write directly to the video memory, or to VGA registers, for performance reasons. (Program access to the display, or to any hardware device, via DOS or BIOS calls can be virtualized, since the DOS or BIOS call is serviced in V86 mode and can be intercepted by DESQview). Protected-mode programs that write directly to the display must be run in full-screen mode. Other tasks can be run simultaneously, but they must run in the background and not in a window on the display.

Resolving contention for the keyboard and mouse is usually not a problem—almost all extended DOS application programs use the real-mode mouse driver and the BIOS to perform I/O to those devices, and these are accessed through software interrupts that can be monitored by DESQview. Likewise, disk I/O is nearly always performed with DOS system calls, which are handled in V86 mode where they can be virtualized.

A significant percentage of extended DOS applications perform I/O directly to the serial communication ports. This can cause conflicts—but even in V86 mode where I/O can be virtualized it's hard to envision a situation where two programs can productively perform I/O over the same comm port. The same is true of most special-purpose hardware that can be added to a PC—it usually only makes sense for one program to use it at a time.

It is possible to imagine pathological cases that could not be run in protected mode under DESQview. For example, if a protected-mode application programs the keyboard controller chip to turn off the keyboard, you probably would not want to run it in a DESQview window. But such socially unacceptable behavior is the exception rather than the rule—practically speaking, the only restriction on multitasking protected-mode programs under DESQview is that they must be run in full-screen mode if they write directly to the display.

Multitasking Two Protected-Mode Programs

It's instructive to consider what actually happens when DESQview multitasks two extended DOS applications. Let's suppose the user opens window A and window B, and wants to run an extended DOS application simultaneously in each window. Figure 10-1 shows how control is passed between DESQview and the two applications. DESQview and QEMM set up a V86 mode environment in which DOS and other real-mode code can run, and a protected-mode environment which is used by DESQview and QEMM for performing system-level activities. Each DOS extender has real-mode code that runs in the V86 environment, and protected-mode code that runs in the protected-mode environment set up by the DOS extender. VCPI calls are used by the DOS extender to switch between V86 mode and its own protected-mode environment.

All communication between the programs is performed in V86 mode, which is the environment they all have in common. The control flow between programs, shown as arrows in Figure 10-1, is accomplished primarily with interrupts—either hardware interrupts, or software interrupts such as DOS and BIOS system calls. The direction of the arrows shows the flow of control when the interrupt occurs; as with any interrupt, control is returned along the same path after the interrupt handler completes.

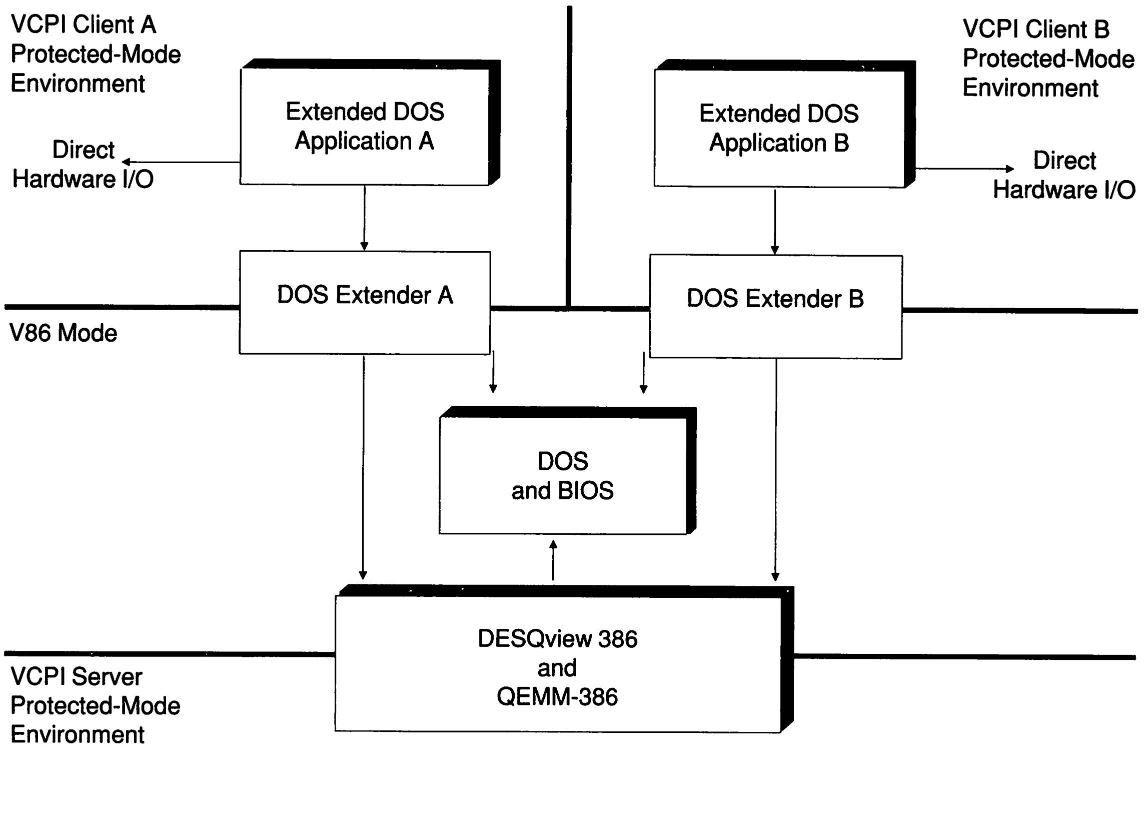
Suppose application A is fired up first. DOS extender A starts running in V86 mode, then uses the VCPI services provided by QEMM to switch to its protected-mode environment, where it loads and runs application A. As hardware interrupts occur, or as DOS or BIOS calls are made by the application, the DOS extender fields them and uses the VCPI services to switch back to V86 mode, where it reissues the interrupt to pass control to the real-mode handler. Most of these interrupts are processed directly by DOS or the BIOS. However, some are handled entirely, or at least filtered, by DESQview and QEMM. For example, DESQview filters the timer tick hardware interrupt so it can monitor elapsed time and switch tasks when a program's time slice has expired.

Eventually the user presses the DESQview hot key and instructs DESQview to start up application B. DOS extender B runs application B in protected mode exactly as described for application A, with the important difference that DOS extender B provides a protected-mode environment that is completely disjoint from DOS extender A's protected-mode environment. After application B has run for a while, its time slice expires. A timer tick interrupt occurs, and DOS extender B switches to V86 mode and reissues the interrupt. DESQview gets control and decides to switch control back to application A. It "returns" to DOS extender A from the keyboard interrupt for the hot key, which was the last event seen by DOS extender A before the user started application B running.

Thus, each application runs until it terminates, gives up control implicitly by execution of a software interrupt for a DOS or BIOS service, its time slice expires, or some other hardware interrupt (including those caused by keyboard entries) occurs. Any of these events can result in DESQview transferring control of the machine from one application to the other. At any given moment, code is running in one of four environments: the V86 mode environment controlled by DESQview/QEMM, DOS extender A's protected-mode environment, DOS extender B's protected-mode environment, or DESQview/QEMM's protected-mode environment.

This seems complex on the surface, but an underlying simplicity is actually present. At any moment, only one protected-mode environment is active. The program controlling that environment—either DESQview/QEMM, DOS extender A, or DOS extender B—can act as if it controls the machine, and as if no other protected mode environment exists. The VCPI interface is used to switch between environments as necessary. Each DOS extender has no knowledge of the other DOS extender, or of DESQview; as far as it's concerned, it's just using the VCPI interface to switch between V86 mode and its own protected-mode environment.

Figure 10-1: Multitasking extended DOS applications with VCPI.


VCPI Client A Protected-Mode Environment
Direct Hardware I/O
Extended DOS Application A
Extended DOS Application B
VCPI Client B Protected-Mode Environment
Direct Hardware I/O
DOS Extender A
DOS Extender B
V86 Mode
DOS and BIOS
DESQview 386 and QEMM-386
VCPI Server Protected-Mode Environment

DESQview doesn't need to treat the extended DOS applications any differently from normal DOS applications; it simply passes control to each program in V86 mode, and the fact that the program later uses the VCPI interface to switch to its own protected-mode environment has no relevance to the multitasking aspect of DESQview. All this environment switching activity can occur without noticeable overhead because the switching is accomplished with a very small piece of code, as was demonstrated in Chapter 8.

As noted previously, the primary problem with the VCPI solution is the inability of the multitasker to virtualize hardware access. This is depicted in Figure 10-1 by the arrows at the top of the diagram showing the extended DOS applications performing direct hardware I/O. Since that I/O is performed in the context of a protected-mode environment controlled by a DOS extender, DESQview cannot make any provision to trap the I/O and virtualize it.

Summary

Compatibility between multitaskers and DOS extenders is an as yet unresolved problem that is currently receiving a lot of attention from developers. The two main solutions being seriously considered by developers are: 1) providing the VCPI interface in the multitasker, and 2) defining a new interface to be provided by the multitasker and used by the DOS extender.

VCPI support in the multitasker is a good short-term answer that yields immediate compatibility with existing extended DOS applications. The VCPI solution is available today in DESQview 386. However, VCPI does not permit the multitasker to virtualize direct hardware access by an extended DOS application. This results in restrictions on how applications can run under the multitasker, notably the requirement that an application which performs direct screen I/O must run in full-screen mode, rather than in a window on the display.

In the longer term, an interface that makes hardware virtualization possible is highly desirable. The DPMI interface provided in Windows 3.0 is one possibility. DPMI may well become an industry standard, if DOS extender vendors choose to implement DPMI support in their products. However, even if DPMI is universally adopted, extended DOS applications capable of using the DPMI interface are still some time away from market. This is due to the time required for a DOS extender vendors to re-engineer their products to add DPMI support, and for applications developers to re-release their products after incorporating a DOS extender with DPMI support.

Vendor Guide

Documents and Specifications

The VCPI interface is available from Phar Lap Software, Inc., 60 Aberdeen Avenue, Cambridge, MA 02138, (617) 661-1510, or Quarterdeck Office Systems, 150 Pico Boulevard, Santa Monica, CA 90405, (213) 392-9851.

Extended

The eXtended Memory Specification (XMS) version 2.0 is available from Microsoft Corporation, Box 97017, Redmond, WA 98073, (206) 882-8080.

The Lotus/Intel/Microsoft Expanded Memory Specification (EMS) version 4.0 is available from Intel Corporation, 5200 N. E. Elam Young Parkway, Hillsboro, OR 97124.

The DESQview API Toolkit is available from Quarterdeck Office Systems, 150 Pico Boulevard, Santa Monica, CA 90405, (213) 392-9851.

Memory Managers and EMS Emulators

386-to-the-Max Professional version 4.07 (386$^{\text{MAX}}$) is available from Qualitas, Inc., 7101 Wisconsin Avenue, Suite 1386, Bethesda, MD 20814, (301) 907-6700.

QEMM-386 is available from Quarterdeck Office Systems, 150 Pico Boulevard, Santa Monica, CA 90405, (213) 392-9851.

CEMM-386 is a product of Compaq Computer Corp., 20555 SH249, Houston, TX 77070, (800) 231-0900.

Turbo EMS is a product of Merrill & Bryan Enterprises, Inc., 9770 Carroll Center Road, Suite C, San Diego, CA 92126, (619) 689-8611.

The source code and executable driver for HIMEM.SYS, Microsoft's implementation of the XMS, is available for downloading from several sources, including the Microsoft SIG on CompuServe.

Multitasking Environments and DOS-compatible Multitasking Operating Systems

DESQview 386 is a product of Quarterdeck Office Systems, 150 Pico Boulevard, Santa Monica, CA 90405 (213) 392-9851.

Windows is a product of Microsoft Corporation, Box 97017, Redmond, WA 98073, (206) 882-8080.

VM/386 is a product of Intelligent Graphics Corporation (IGC), 4800 Great America Parkway, Santa Clara, CA 95054, (408) 986-1431.

PC-MOS/386 is a product of The Software Link, 3577 Parkway Lane, Norcross, GA 30092, (404) 448-5465.

Concurrent DOS 386 is a product of Digital Research Inc., 70 Garden Court, Monterey, CA 93940, (800) 443-4200.

DOS Extenders

386 | DOS-Extender and 386 | VMM are available from Phar Lap Software, Inc., 60 Aberdeen Avenue, Cambridge, MA 02138, (617) 661-1510.

OS/286, OS/386, and OS/386-HB are available from Eclipse Computer Solutions, Inc., One Intercontinental Way, Peabody, MA 01960, (508) 535-7510.

DOS/16M is available from Rational Systems, Inc., 220 North Main St., Natick, MA 01760, (508) 653-6006.

X-AM was developed by Intelligent Graphics Corporation, 4800 Great America Parkway, Santa Clara, CA 95054, (408) 986-1431, but is not currently being marketed.

OTG Systems, P. O. Box 239, Clifford, PA 18413 (717-222-9100), markets a 386 DOS Extender that is only supported by their Fortran FTN77/386 development system.

Extended DOS Programming Tools

Alsys, Inc. (Ada)	Catspaw (SPITBOL)	Digitalk, Inc. (Smalltalk)
67 South Bedford St.	P. O. Box 1123	9841 Airport Blvd.
Burlington, MA 01803	Salida, CO 81201	Los Angeles, CA 90045
(617) 270-0030	(719) 539-3884	(213) 645-1082

Epsilon (Prolog)
Kurfurstendamm 188, D-1000
Berlin 15, W. Germany
FAX: (49) 30 88 23 594

Expert Systems (Prolog)
7 W. Way, Unit 14
Botley, Oxford OX2-OJB
England
FAX: (44) 86 52 50 270

INTEK (C++)
1400 112th Avenue, SE
Bellevue, WA 98004
(208) 455-9935

Laboratory Microsystems, Inc.
(Forth)
12555 West Jefferson Blvd., Suite 202
Los Angeles, CA 90066
(213) 306-7412

Lahey Computer Systems, Inc.
(FORTRAN)
P. O. Box 6091
Incline Village, NV 89450
(800) 548-4778

Language Processors, Inc. (LPI)
(BASIC, COBOL, FORTRAN, PL/I)
959 Concord Street
Framingham, MA 01701
(508) 626-0006

Lattice (C)
2500 South Highland Avenue
Lombard, IL 60148
(800) 444-4309

MetaWare Inc. (C, Pascal)
2161 Delaware Ave
Santa Cruz, CA 95060
(408) 429-6382

Microsoft Corporation (C,
Windows SDK)
Box 97017
Redmond, WA 98073
(206) 882-8080

MicroWay, Inc. (C, C++, FOR-
TRAN, Pascal)
Cordage Park, Building 20
Plymouth, MA 02360
(508) 746-7341

OASYS (C, C++, FORTRAN,
Pascal)
230 Second Avenue
Waltham, MA 02154
(617) 890-7889

OTG Systems, Inc. (FORTRAN)
P. O. Box 239
Clifford, PA 18413
(717) 222-9100

ParcPlace Systems (Smalltalk)
1550 Plymouth Street
Palo Alto, CA 94043
(800) 822-7880

Phar Lap Software
(386 | ASM/LINK, LinkLoc)
60 Aberdeen Avenue
Cambridge, MA 02138
(617) 661-1510

Quarterdeck Office Systems
(DESQview API Toolkit)
150 Pico Boulevard
Santa Monica, CA 90405
(213) 392-9851

Rational Systems, Inc. (C)
220 North Main St.
Natick, MA 01760
(508) 653-6006

R. R. Software, Inc. (Ada)
2317 International Lane,
Suite 212
Madison, WI 53704
(608) 244-6436

Silicon Valley Software (BASIC,
C, FORTRAN, Pascal)
1710 South Amphlett Boulevard,
Suite 100
San Mateo, CA 94402
(415) 572-8800

STSC, Inc. (APL)
2115 East Jefferson Street
Rockville, MD 20852
(301) 984-5123

Symbolics (LISP)
#8 New England Executive Park
Burlington, MA 01803
(617) 621-7500

Telesoft (Ada)
5959 Cornerstone Court West
San Diego, CA 92121
(619) 457-2700

TransEra Corporation (BASIC)
3707 North Canyon Road
Provo, UT 84604
(801) 224-6550

Watcom Systems, Inc. (C)
415 Phillip Street
Waterloo, Ontario
Canada N2L 3X2
(519) 886-3700

Whitewater Group (Actor)
600 Davis Street
Evanston, IL 60201
(708) 328-9386

Index

IBM Programming

A ▶ BENCHMARK BOOK

> $22.95 FPT USA

Maximize Program Performance with Protected-Mode Programming, Multitasking, and More Memory

Extending DOS will help bring all the power of a new generation of PC hardware to your DOS applications. Written by Ray Duncan and an all-star team of authors, it is a complete and authoritative guide to all aspects of extending and enhancing DOS. It will show you how to address memory above 640K with EMS and XMS standards, take advantage of the speed and increased addressable memory of 286 and 386 architecture, understand industry standards such as VCPI and DPMI, and implement operating environments such as Windows and DESQview.

Extending DOS gives you the unique perspectives of the most respected and qualified DOS programming authors, each writing about his own area of expertise. The contents include:

- IBM PC Programming Architecture
- Expanded Memory Specification (EMS)
- eXtended Memory Specification (XMS)
- 286 and 386 Extenders
- Windows Operating Environment
- DESQview Operating Environment
- VCPI and DPMI

ABOUT THE AUTHORS

Ray Duncan is a contributing editor to *PC Magazine* and is the author of *Advanced MS-DOS Programming* and *Advanced OS/2 Programming* as well as numerous other books.

Charles Petzold writes for *PC Magazine* and *Microsoft Systems Journal*. His books include *Programming Windows* and *Programming the OS/2 Presentation Manager.*

M. Steven Baker is the editor of *Programmer's Journal*. He writes frequently on 80386 development.

Andrew Schulman is a software engineer and contributing editor to *Dr. Dobb's Journal*. He has also written for *BYTE* and *Microsoft Systems Journal*.

Stephen R. Davis is the author of *DESQview: A Guide to Programming the DESQview Multitasking Environment*, among other titles.

Ross P. Nelson is Manager of Software Engineering at Answer Software and is the author of *The 80386 Book* as well as articles for *BYTE* and *Dr. Dobb's Journal*.

Robert Moote is a cofounder and vice president of Phar Lap Software, Inc., the creator of Phar Lap's 386 | DOS-Extender and 386 | VMM products.

Cover design by Copenhaver Cumpston

Addison-Wesley Publishing Company, Inc.

52295
9 780201 550535
ISBN 0-201-55053-9
55053